Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jani Bevk

Določanje soodvisnih sprememb proteinov z uporabo grafičnih procesnih enot

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič
Somentor: doc. dr. Tomaž Curk

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Na osnovi opazovanja hkratnega spreminjanja delov proteinov tekom evolucije je moč sklepati tako o funkcionalni pomembnosti posameznih delov kakor tudi o interakcijah med pari proteinov. Zaradi obsežnosti genomov, ki lahko vsebujejo več desettisoč genov, je to računsko in statistično zahtevno opravilo. Preučite algoritme za detekcijo soodvisnih sprememb proteinov. Izbrani algoritem prilagodite za grafične procesne enote in poročajte o doseženih pohitritvah in energijski učinkovitosti glede na izvedbo za centralne procesne enote.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Jani Bevk sem avtor diplomskega dela z naslovom:

Določanje soodvisnih sprememb proteinov z uporabo grafičnih procesnih enot

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Uroša Lotriča in somentorstvom doc. dr. Tomaža Curka,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 2. februarja 2015 Podpis avtorja:

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DNK	Deoxyribonucleic acid	Deoksiribonukleinska kislina
RNK	Ribonucleic acid	Ribonukleinska kislina
MSA	Multiple Sequence Alignment	Poravnava več zaporedij
GPU	Graphics processing unit	Grafična procesna enota
CPU	Central processing unit	Centralna procesna enota
GPGPU	General-purpose computing on graphics processing units	Splošno-namensko računanje na grafičnih procesnih enotah
CUDA	Compute Unified Device Architecture	Platforma za vzporedno programiranje na grafičnih procesnih enotah

Povzetek

Proteini sodelujejo v skoraj vseh procesih, ki se odvijajo v celicah živečih organizmov. Razumevanje njihovih funkcij je ključnega pomena za razumevanje bioloških procesov. Soodvisne spremembe v proteinih, ki se tekom njihove evolucije odvijajo, so tesno povezane z njihovo strukturo in funkcijo. Razvitih je bilo več različnih algoritmov za detekcijo soodvisnih sprememb, vsi pa so računsko zelo zahtevni. Eden izmed takšnih algoritmov je algoritem OMES, ki temelji na statističnem testu χ^2 in naključnem mešanju poravnav proteinskih zaporedij. Cilj diplomske naloge je paralelizacija in implementacija algoritma OMES na grafični procesni enoti z uporabo platforme CUDA. Grafične procesne enote so specializirani procesorji, ki danes najverjetneje nudijo najboljše razmerje med računsko močjo in ceno. V primerjavi z implementacijo na centralni procesni enoti smo dosegli stokratne pohitritve in porabili manj energije.

Ključne besede: analiza soodvisnih sprememb proteinov, OMES, splošnonamensko računanje na grafičnih procesnih enotah, CUDA.

Abstract

Proteins are involved in almost all processes that take part in cells of living organisms. Understanding of their function is important for the understanding of biological processes. Correlated mutations in proteins, which take place over the course of their evolution, are closely related to their structure and function. Several different algorithms for detection of correlated mutations have been developed, all very computationally intensive. One of such algorithms is the OMES algorithm, which is based on the statistical χ^2-test and random shuffling of protein sequences. The aim of the thesis is paralelisation and implementation of the OMES algorithm on graphics processing units using the CUDA platform. Graphics processing units are specialized processors, which nowadays probably offer the best computing power to price ratio. Compared to the implementation on a central processing unit we achieved a 100-fold speedup and used less energy.

Keywords: correlated mutation analysis, OMES, general-purpose computing on graphics processing units, CUDA.

Poglavje 1

Uvod

V zadnjih desetletjih so računalniki postali nepogrešljiv del našega vsakdanjega življenja. Nepogrešljivi so postali tudi v moderni znanosti. Znanstveniki jih s pridom uporabljajo za reševanje problemov iz najrazličnejših področij. A za reševanje čedalje večjih in kompleksnejših problemov seveda potrebujemo vedno bolj zmogljive računalnike. Običajno zmogljivost računalnika povezujemo s frekvenco centralne procesne enote. Opazimo lahko, da ta v zadnjih letih narašča čedalje počasneje oziroma se skoraj ustavlja. Razlog tiči v tem, da ne znamo dovolj učinkovito odvajati toplote, ki se sprošča pri visokih frekvencah. Zato razvoj procesorjev sedaj stremi k večjedrnosti in paralelnosti, ki omogoča sočasno izvajanje večih ukazov. Tu pa procesorje povsem zasenčijo grafične procesne enote, ki na več sto procesnih jedrih izvajajo več deset tisoč vzporednih niti.

Grafične procesne enote so specializirani procesorji, namenjeni prikazovanju računalniške grafike. Na današnjih grafičnih karticah so se, predvsem po zaslugi industrije računalniških iger, razvile v izjemno zmogljive in prilagodljive procesorje. Danes najverjetneje nudijo največje razmerje med računsko močjo in ceno. Čeprav so bile grafične procesne enote sprva namenjene zgolj računalniški grafiki, pa jih lahko uporabljamo tudi za računanje, nepovezano z grafiko. Izid platforme CUDA je povzročil velik val zanimanja za izkoriščanje grafičnih procesnih enot za znanstveno računanje, med drugim

tudi na področjih računske biologije in bioinformatike.

Molekularna koevolucija [1] je eno izmed najbolj aktivnih področij raziskav v bioinformatiki. Analiza soodvisnih mutacij je način za predvidevanje strukture proteinov in interakcij med njimi. Proteini igrajo veliko vlogo v našem življenju, saj opravljajo pomembne naloge v vseh celicah živečih organizmov. Razumevanje funkcij proteinov je ključnega pomena za razumevanje bioloških procesov in posledično tudi za razvoj novih zdravil. Kljub izboljšavam eksperimentalnih postopkov za določitev strukture proteinov se razkorak med številom znanih proteinskih zaporedij in poznavanjem njihovih struktur še naprej veča [2]. Zato ni prav nič čudno, da se je razvila cela kopica računskih metod za detekcijo molekularne koevolucije in predvidevanje struktur proteinov. Vsem pa je skupno to, da so računsko precej zahtevne.

Zato bomo v tem diplomskem delu paralelizirali in poizkusili pohitriti enega od algoritmov za detekcijo soodvisnih mutacij. Izbrali smo algoritem OMES (*Observed Minus Expected Squared*), ki temelji na statističnem testu χ^2. Zaradi svoje velike podatkovne paralelnosti je kot nalašč primeren za implementacijo na grafični procesni enoti. Algoritem bomo implementirali tako na sekvenčni, tradicionalen način, kot tudi z uporabo platforme CUDA. Na koncu bomo primerjali čase, ki jih implementaciji potrebujeta za izračun rezultata pri različnih velikostih vhodnih podatkov in izračunali pohitritev, ki smo jo dosegli z implementacijo algoritma na grafični procesni enoti. Preverili bomo tudi kakšna je poraba energije med izvajanjem obeh implementacij.

Poglavje 2

Koevolucija in analiza soodvisnih mutacij

2.1 Koevolucija

Koevolucija igra zelo pomembno vlogo v ključnih bioloških sistemih. Je posebna vrsta evolucijskega procesa, pri katerem se dve različni vrsti prilagajata evolucijskim spremembam druga druge, in je odgovorna za velik del bogate biološke raznovrstnosti na našem planetu.

Najbolj razširjena definicija koevolucije je Thompsonova, ki koevolucijo strogo definirana kot vzajemno evolucijsko spreminjanje v vrstah, ki medsebojno vplivajo druga na drugo [3]. Po tej definiciji so koevolucijske spremembe samo tiste, ki nastanejo pod vplivom biotskih dejavnikov, to je interakcij med sobivajočimi organizmi. Spremembe, ki so posledica abiotskih dejavnikov (na primer spremembe v okolju), niso koevolucijske. Pomemben del definicije je vzajemnost sprememb [1]. Evolucijska sprememba v eni vrsti skozi selekcijski pritisk povzroči spremembo v drugi vrsti. Kot odgovor na to spremembo lahko druga vrsta povzroči vzajemno spremembo nazaj v prvi vrsti. Koevolucija je torej tesno povezana z naravno selekcijo. Ker preživijo le najuspešnejši posamezniki, se v naslednje generacije vrste prenesejo le tiste spremembe, ki so bile dejansko koristne.

Do koevolucije lahko po [4] pride pri naslednjih tipih interakcij med vrstami:

1. **Odnos plenilec-plen:**
   Evolucija plena vpliva na evolucijo plenilca in obratno. Plenilec teži k razvoju lastnosti, ki mu olajšajo lov (na primer čutila, kremplji), plen pa k razvoju lastnosti, ki mu pomagajo pri izmikanju plenilcem (na primer kamuflaža, hitrost). Govorimo o evolucijski oboroževalni tekmi.

2. **Odnos gostitelj-zajedavec:**
   Odnos je precej podoben odnosu plenilec-plen. Pomembna razlika je, da je zajedavec odvisen od gostitelja ne samo za hrano, pač pa tudi za zavetje in razmnoževanje. Hitra smrt gostitelja ni v interesu zajedavca.

3. **Mutualizem in priskledništvo:**
   Dve vrsti se evolucijsko razvijeta tako, da med seboj sodelujeta. To koristi eni (priskledništvo) ali obema (mutualizem) vrstama. Tipičen primer mutualizma je odnos med čebelami in rožami.

4. **Tekmovanje:**
   Več vrst med seboj tekmuje za prevlado (na primer prostor, svetlobo, hrano) in preživetje.

Vendar pa vsaka takšna interakcija med dvema vrstama ni nujno povezana s koevolucijo [5]. Na primer, prisotnost zajedavca na gostitelju še ni nujno dokaz, da se med zajedavcem in gostiteljem odvija proces koevolucije. Zajedavec je namreč lahko imel lastnosti, ki mu omogočajo zaobiti gostiteljeve obrambe, že preden je začel napad na gostitelja. Da bi se prepričali, če je med vrstama res prišlo do koevolucije, moramo poiskati in dokumentirati vzajemne spremembe in izvesti filogenetsko analizo[1].

[1] Filogenetika je veda, ki se ukvarja z evolucijskimi sorodstvi med skupinami organizmov (geni, populacijami, vrstami) [6]. S filogenetsko analizo na podlagi sekvenciranja molekul sklepamo o teh sorodstvih.

2.2 Molekularna koevolucija

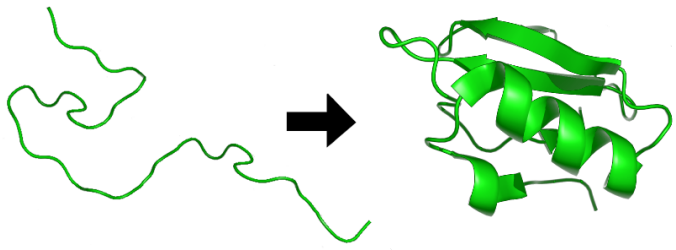
Koevolucija igra pomembno vlogo na vseh bioloških nivojih. Opazimo jo lahko tako na makroskopskem nivoju, kjer vpliva na kovariiranje značilnosti med različnimi vrstami (poglavje 2.1), kot tudi na mikroskopskem nivoju, kjer govorimo o molekularni koevoluciji. Koevolucija na nivoju molekul je povsem analogna koevoluciji na nivoju vrst, le da se vzajemne spremembe odvijajo med molekulami. Lovell in Robertson sta v [1] prilagodila Thompsonovo definicijo koevolucije (poglavje 2.1) in definirala molekularno koevolucijo kot vzajemno evolucijsko spreminjanje lokusov, ki medsebojno evolucijsko vplivajo drug na drugega. Lokus je specifična lokacija gena (ali pomembne DNA sekvence) na kromosomu [7].

Molekularna koevolucija se lahko odvija tako znotraj iste molekule (intramolekularna koevolucija), kot tudi med različnimi molekulami (intermolekularna koevolucija) [1]. Najbolj znani primeri molekularne koevolucije so spremembe v zaporedjih nukleotidov v deoksiribonukleinskih kislinah (DNA) in ribonukleinskih kislinah (RNA) ter spremembe v zaporedjih aminokislinskih ostankov proteinov. Bolj specifično, sprememba na nekem mestu v zaporedju aminokislinskih ostankov v enem proteinu lahko povzroči vzajemno spremembo na nekem drugem mestu v istem proteinu ali pa spremembo na nekem mestu v drugem proteinu. Razlog za nastanek druge spremembe je ohranjanje funkcije, strukture in/ali stabilnosti proteina, ki jih je lahko ogrozila prva sprememba [8]. To lahko interpretiramo tudi kot izvajanje selektivnega pritiska prvega mesta spremembe na drugo mesto.

Da lahko proteini opravljajo svojo funkcijo imajo kompleksno tridimenzionalno strukturo in so v interakciji z drugimi proteini in molekulami. Povsem pričakovano je, da soodvisne spremembe nastanejo na mestih, ki so zaradi strukture in interakcij fizično v bližini. Znani pa so tudi primeri, ko se soodvisne spremembe pojavijo med proteini, ki niso v direktni interakciji drug z drugim. Spremembe se namreč lahko propagirajo tudi med bolj oddaljenimi proteini [9].

2.2.1 Proteini

Proteini so velike, kompleksne molekule, ki opravljajo veliko pomembnih funkcij v živečih organizmih. Sestavljeni so iz ene ali več verig aminokislinskih ostankov[2], ki so med seboj povezani s peptidno vezjo. Verige so sestavljene iz ostankov dvajsetih različnih standardnih aminokislin [10]. Proteini se med seboj razlikujejo v dolžini verig (običajno med sto in tisoč aminokislinskimi ostanki) in vrstnem redu aminokislinskih ostankov v verigah. Zaporedju aminokislinskih ostankov rečemo primarna zgradba proteina. Vrstni red aminokislinskih ostankov je določen z zaporedjem nukleinskih kislin v genu.



Slika 2.1: Primer zvitja proteina v kompleksno tridimenzionalno strukturo. Povzeto po [11].

Vrstni red aminokislinskih ostankov določa unikatno tridimenzionalno strukturo vsakega proteina (slika 2.1). Zaradi interakcij med posameznimi deli se veriga aminokislinskih ostankov zvije v kompleksno tridimenzionalno strukturo. Tridimenzionalna struktura proteina točno določa kakšno funkcijo opravlja protein. Ta posebna struktura mu namreč omogoča interakcijo z drugimi proteini in molekulami.

Proteini so bistvenega pomena za vse celice. Sodelujejo pri skoraj vseh procesih, ki se v njih odvijajo. Njihove funkcije med drugim vključujejo:

- pospeševanje kemičnih reakcij,
- sodelovanje pri metabolizmu,

[2] Izraz aminokislinski *ostanek* se uporablja zato, ker se pri nastanku peptidne vezi od aminokisline odcepi molekula vode. Verigo torej sestavljajo le ostanki aminokislin.

- prenos signalov med celicami,
- delitev celic,
- transport,
- zagotavljanje strukturne podpore, in
- imunske odzive.

Razumevanje strukture proteinov in interakcij med njimi je torej nadvse pomembno za razumevanje funkcij proteinov in načina delovanja celičnih sistemov in organizmov [12]. Žal pa je določanje tridimenzionalne strukture proteinov z biološkimi metodami precej nepraktično in drago [2]. Zato sta računsko predvidevanje tridimenzionalne strukture proteinov in predvidevanje interakcij med proteini dve pomembnejši področji v bioinformatiki.

2.2.2 Poravnava več zaporedij

Poravnava več zaporedij (angl. *Multiple Sequence Alignment, MSA*) je poravnava več proteinskih zaporedij aminokislinskih ostankov (lahko pa tudi zaporedij nukleotidov v DNA ali RNA) v pravokotno matriko, običajno tako, da je vsako zaporedje v svoji vrstici. V večini primerov so ta zaporedja *homologna*, kar pomeni, da so domnevno evolucijsko povezana in imajo skupne prednike. Vsako zaporedje lahko predstavimo kot niz črk, kjer vsaka črka predstavlja točno določen element zaporedja. Na primer, črka A predstavlja aminokislinski ostanek alanin. Cilj je najti takšno razporeditev, da so si aminokislinski ostanki v danem stolpcu enaki ali čim bolj podobni. Da najdemo čim boljše ujemanje, v zaporedja vstavimo vrzeli. Primer takšne matrike lahko vidimo na sliki 2.2.

```
S119  MKVIYIYLLLLLVCKFLFVRSSCSLKVGKIECTNEFETFLLYNETNVNVKMDTVG
S202  MKVIYIYLLLLLVCKFLFVRSSCSLKVGKIECTNEFETFLLYNETNVNVKMDTVG
MA12  MKVIYIYLLLLLVCKFLFVKSSCSLENEKIECTNELETFLLYNETVVNVKMDKNG
NC4   MKVIYIYLLLLLVCKFLFVKSSCSLKVGKIECTKELETFLLYNETVVNVKMDTNG
NC472 MKVIYIYLLLLLVCKFLFVKSSCTLKVGKIECTNEFETFELYNGTVVDVKMDTVG
JN512 MKVI--YVLILLVCKFLFVKSSCYFQGKNLQCSKEFEKIELYNGTEVNVKMDTVG
```

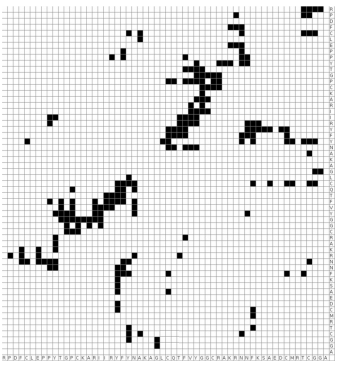
Slika 2.2: Primer poravnave 6 zaporedij dolžine 55 aminokislinskih ostankov.

V nastali matriki lahko identificiramo regije podobnih zaporedij, ki so posledica evolucijskih, strukturnih ali funkcijskih razmerij med zaporedji. Če si zaporedja delijo skupne prednike, lahko neujemanja interpretiramo kot mutacije. Nasprotno, odsotnost neujemanj v določeni regiji pa lahko pomeni, da je regija funkcijsko ali strukturno pomembna za protein. Poravnave več zaporedij so torej osnova za predvidevanje strukture in funkcije proteinov, analizo soodvisnih mutacij, filogenetsko analizo in še veliko drugih pogostih postopkov pri analizi zaporedij.

Razporejanje zaporedij v matriko ni trivialen postopek. Obstaja več različnih metod, vse pa so zelo računsko zahtevne [13, 14].

2.2.3 Karta proteinskih stikov

Karta proteinskih stikov (angl. *protein contact map*) je dvodimenzionalna reprezentacija kompleksne tridimenzionalne strukture proteina. Je matrika (slika 2.3), katere red je enak številu aminokislinskih ostankov v proteinu. Element matrike (i, j) predstavlja razdaljo med aminokislinskima ostankoma na mestih i in j. Največkrat je to binarna vrednost, kjer vrednost 1 predstavlja kontakt med aminokislinskima ostankoma. Aminokislinska ostanka sta v kontaktu, če je razdalja med njima manjša ali enaka določeni mejni vrednosti. Matriko zlahka razširimo tako, da prikazuje razdalje med aminokislinskimi ostanki dveh različnih proteinov.



Slika 2.3: Primer binarne karte proteinskih stikov [15].

Karta proteinskih stikov je v bioinformatiki zelo pogosto uporabljen način za predstavitev kontaktov med aminokislinskimi ostanki. Uporabljamo jih lahko za primerjavo, napovedovanje in vizualizacijo struktur proteinov. Njihova prednost pred tridimenzionalnimi modeli proteinov sta predvsem neobčutljivost na rotacije in translacije proteinov.

2.3 Analiza soodvisnih mutacij

Analiza soodvisnih mutacij (angl. *Correlated Mutation Analysis, CMA*) je način za detekcijo molekularne koevolucije in napovedovanje interakcij proteinov. Cilj teh metod je identifikacija mest parov aminokislinskih ostankov, ki soodvisno mutirajo pogosteje, kot bi to lahko pričakovali od aminokislinskih ostankov, ki niso v stiku drug z drugim. V ta namen je bilo razvitih veliko različnih računskih metod, a so bile omejene z algoritmičnimi omejitvami in visoko računsko zahtevnostjo. Šele nedavni napredki v razvoju algoritmov (na primer zmanjševanje evolucijskega šuma) in računski moči so omogočili bistveno napredovanje pri iskanju soodvisnih mutacij [16].

Koevolucija se sicer lahko odvija med različnimi vrstami molekul, vendar se najnovejša orodja in metode osredotočajo na koevolucijo proteinov. Te metode lahko v grobem razdelimo v dve skupini. Prve delujejo na nivoju aminokislinskih ostankov, druge pa na nivoju proteinov [17].

Večina metod, ki deluje na nivoju aminokislinskih ostankov za izhodišče uporablja poravnave več zaporedij (poglavje 2.2.2). Delujejo tako, da v poravnavah identificirajo mesta, na katerih med različnimi zaporedji lahko opazimo soodvisne spremembe. Ta mesta pa ustrezajo dejanskim mestom v verigi aminokislinskih ostankov v proteinih. Detekcija koevolucije aminokislinskih ostankov ponavadi poteka v dveh korakih [18]:

1. konstrukcija oziroma pridobitev poravnave več zaporedij proteina (ali družine proteinov), in
2. izračun številčne ocene soodvisnosti za vsak par mest v tej poravnavi.

Metode se med seboj razlikujejo predvsem v drugem koraku, saj vsaka izračuna številčno oceno na drugačen način.

Metode, ki delujejo na nivoju proteinov, za izhodišče uporabljajo filogenetska drevesa družin proteinov. Filogenetska drevesa so razvejani diagrami, ki prikazujejo domnevna evolucijska razmerja med biološkimi vrstami, organizmi ali geni. Ideja metod je v tem, da molekularno koevolucijo, ki jo povzročijo interakcije med proteini, lahko prepoznamo kot podobnosti v pripadajočih filogenetskih drevesih proteinov. Podobnosti namreč lahko opazimo

tudi na makroskopskem nivoju med filogenetskimi drevesi vrst, ki sodelujejo v koevoluciji.

Z analizo soodvisnih mutacij lahko torej identificiramo mesta aminokislinskih ostankov, ki sodelujejo v koevoluciji. Rezultate največkrat prikažemo v obliki karte proteinskih stikov (poglavje 2.2.3). Z njo lahko predvidimo tridimenzionalno strukturo proteinov, identificiramo aktivna mesta na proteinu in napovemo interakcije med proteini. To nam omogoča boljše razumevanje funkcije proteinov in proteinskih kompleksov ter pripomore k večanju znanja o celicah, celičnih sistemih in organizmih.

Zato ni prav nič čudno, da se je skozi leta pojavilo veliko različnih metod za detekcijo soodvisnih mutacij [17, 19]. V nadaljevanju poglavja bomo pregledali zgolj najbolj priljubljene metode, ki soodvisne mutacije iščejo na nivoju aminokislinskih ostankov.

2.3.1 McBASC

Algoritem McBASC (angl. *McLachlan Based Substitution Correlation*) [20] je eden od najbolj priljubljenih algoritmov za detekcijo soodvisnih mutacij. Od ostalih algoritmov se razlikuje v tem, da poleg poravnav več zaporedij uporablja še matrike nadomeščanja (angl. *substitution matrices*), ki opisujejo stopnje zamenjave ene aminokisline z drugo. V implementaciji se običajno uporabi McLachlanovo matriko nadomeščanja, po kateri je algoritem tudi dobil ime. S pomočjo te matrike se za vsako mesto v poravnavi več zaporedij izračuna številčna ocena, ki jo z uporabo Pearsonovega korelacijskega koeficienta primerjamo z drugimi mesti [16]. Mesta z visoko stopnjo korelacije so soodvisna. Algoritem McBASC velja za enega izmed bolj natančnih algoritmov za predvidevanje strukturnih interakcij [21].

2.3.2 Medsebojna informacija

Ta metoda uporablja za detekcijo soodvisnosti koncept medsebojne informacije (angl. *mutual information*) iz teorije informacij. Medsebojna informacija

nam pove, za koliko se poveča naše znanje o mestu j, če poznamo mesto i [19]. Večja kot je medsebojna informacija, bolj sta mesti soodvisni. Medsebojno informacijo izračunamo po enačbi

$$MI_{i,j} = \sum_x \sum_y P(x,y) \log \frac{P(x,y)}{P(x)P(y)} \quad . \tag{2.1}$$

$P(x)$ in $P(y)$ sta verjetnosti, da najdemo aminokislinski ostanek x na mestu i in aminokislinski ostanek y na mestu j, $P(x,y)$ pa je verjetnost, da najdemo najdemo aminokislinska ostanka x, y na mestih i, j.

Metoda je precej priljubljena predvsem zaradi intuitivne interpretacije, vendar v nasprotju z McBASC ne upošteva podobnosti med aminokislinskimi ostanki.

2.3.3 SCA

Algoritem SCA (angl. *Statistical coupling analysis*) detektira soodvisne mutacije tako, da oceni kako se distribucija aminokislinskih ostankov na nekem mestu v poravnavi več zaporedij spremeni, če na nekem drugem mestu perturbiramo (naključno premešamo) distribucijo aminokislinskih ostankov (obdržimo na primer samo zaporedja z določenim aminokislinskim ostankom). Stopnjo evolucijske odvisnosti med aminokislinskimi ostanki izrazimo v smislu energije, imenovane *statistical coupling energy* [22]. To izračunamo kot razliko med ocenami originalne in perturbirane poravnave. Originalna različica tega algoritma je imela podobno učinkovitost kot metoda medsebojne informacije, vendar so se kmalu pojavile izboljšane različice algoritma in tudi druge metode, osnovane na perturbaciji.

2.3.4 OMES

Algoritem OMES (angl. *Observed Minus Expected Squared*) [9] je algoritem za detekcijo soodvisnih mutacij med aminokislinskimi ostanki. Za izhodišče uporablja poravnave več zaporedij in temelji na Pearsonovem statističnem testu prileganja χ^2 (angl. χ^2 *goodness-of-fit test*). S testom χ^2 za vsak možen

par mest v poravnavi več zaporedij primerja opazovane (dejanske) pojavitve s pričakovanimi pojavitvami aminokislinskih ostankov na teh mestih. Tako identificira tiste pare mest, kjer se določeni ostanki pojavijo pogosteje kot bi lahko pričakovali, če bi bili mesti povsem neodvisni.

Obstaja več različic in izboljšav tega algoritma [9, 21, 23]. Za detekcijo soodvisnih mutacij sta test χ^2 prva uporabila Kass in Horovitz leta 2002 [9]. Algoritem, ki smo ga paralelizirali v tem delu in je bolj podrobno opisan v nadaljevanju poglavja, je povzet po različici algoritma, imenovani detekcija soodvisnih mutacij z naključnim mešanjem (angl. *detection of correlated mutations using random shuffling*) [23].

Algoritem

Najprej za vsako mesto v poravanvi več zaporedij preštejemo kolikokrat se na njem pojavi vsak aminokislinski ostanek. S spremenljivko $f_{A,i}$ označimo delež zaporedij, ki na mestu i vsebujejo aminokislinski ostanek A, s spremenljivko $f_{B,j}$ pa delež zaporedij, ki na mestu j vsebujejo aminokislinski ostanek B. *Pričakovano* število zaporedij, N_{EX}, ki hkrati vsebujejo aminokislinski ostanek A na mestu i in aminokislinski ostanek B na mestu j izračunamo po enačbi

$$N_{EX} = N_{seq} \cdot f_{A,i} \cdot f_{B,j} \quad . \tag{2.2}$$

Pri tem je spremenljivka N_{seq} število vseh zaporedij v poravnavi. Enačba predpostavlja da med mestoma i in j ni soodvisnosti.

Aminokislinske ostanke na mestu j nato velikokrat naključno premešamo (v [23] je teh mešanj 2000), pri čemer pustimo aminokislinske ostanke na mestu i nedotaknjene. Po vsakem mešanju ugotovimo kakšno je *opazovano* število zaporedij, N_{OBS}, ki hkrati vsebujejo aminokislinski ostanek A na mestu i in aminokislinski ostanek B na mestu j. Nato po enačbi

$$\chi^2_r(i,j) = \sum_{k=1}^{n} \frac{(N_{OBS,k} - N_{EX,k})^2}{N_{EX,k}} \tag{2.3}$$

izračunamo vrednost $\chi^2_r(i,j)$ za mesto i in premešano mesto j. Pri tem je spremenljivka r zaporedna številka mešanja, spremenljivka n pa predstavlja

število različnih parov aminokislinskih ostankov, ki jih lahko najdemo na mestih i in j, če je l število različnih aminokislinskih ostankov na mestu i, m pa na mestu j. Velja $n = l \cdot m$. Vrednost $\chi_r^2(i,j)$ izračunamo tudi za mesto i in še *nepremešano* mesto j. Označimo jo s $\chi_0^2(i,j)$.

Po končanem mešanju lahko paru mest i in j priredimo statistično značilnost korelacij, izraženo z vrednostjo p (angl. *p-value*). Določimo jo glede na primerjavo vrednosti $\chi_0^2(i,j)$ z vsemi vrednostmi $\chi_r^2(i,j)$, dobljenimi po vsakem mešanju. Uporabimo enačbi

$$p(i,j) = \frac{\sum_{r=1}^{N} C_r(i,j)}{N} \quad \text{in} \tag{2.4}$$

$$C_r(i,j) = \begin{cases} 1, & \text{če } \chi_r^2(i,j) \geq \chi_0^2(i,j) \\ 0, & \text{sicer} \end{cases} . \tag{2.5}$$

Spremenljivka N predstavlja število mešanj mesta j. Z izrazom v imenovalcu enačbe 2.4 preštejemo koliko vrednosti $\chi_r^2(i,j)$ je bilo večjih ali enakih vrednosti $\chi_0^2(i,j)$.

Vrednost p je definirana kot verjetnost, da dobimo rezultate, ki so enaki ali bolj ekstremni od dejansko opazovanih rezultatov, pri predpostavki, da velja ničelna hipoteza [24]. V kolikor je vrednost p manjša od vnaprej določene kritične vrednosti (ponavadi je to 5% ali 1%), lahko ničelno hipotezo zavrnemo.

V našem primeru je ničelna hipoteza domneva, da v enačbi 2.2 mesti i in j med seboj nista soodvisni. Če je dobljena vrednost p za par mest i in j zelo majhna, lahko z veliko verjetnostjo zaključimo, da predpostavka o neodvisnosti tega para mest ne drži. Postavimo torej lahko alternativno hipotezo, da sta mesti soodvisni.

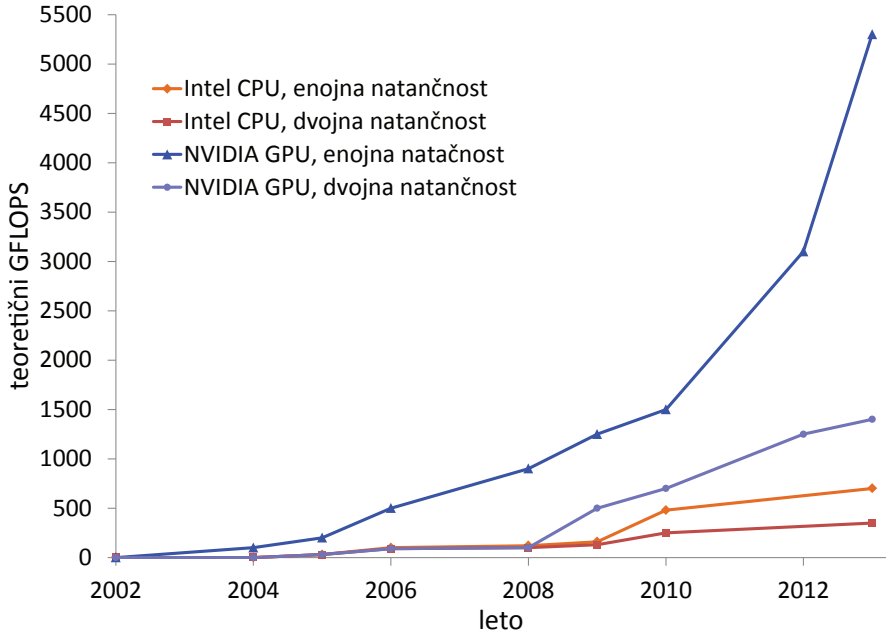
Celoten postopek ponovimo za vse možne pare mest i in j v poravnavi več zaporedij.

Poglavje 3

Splošno-namensko računanje na grafičnih procesnih enotah

Grafična procesna enota (angl. *Graphics processing unit, GPU*) je specializirano integrirano vezje, ki se primarno uporablja pri prikazovanju računalniške grafike. Med drugim to vključuje transformacije objektov pred izrisom, osvetljevanje, senčenje, preslikovanje tekstur in upodobitev poligonov. Včasih so se vse te operacije izvajale na centralnih procesnih enotah (angl. *Central processing unit, CPU*). Z razvojem vse bolj grafično intenzivnih aplikacij so centralne procesne enote postajale preobremenjene in trpela je njihova učinkovitost. V iskanju rešitve, kako razbremeniti centralne procesne enote, so se razvile grafične procesne enote, katerih namen je bil sprva hitrejše prikazovanje računalniške grafike. Sčasoma so se razvile v komponente z večjo računsko močjo in prepustnostjo pomnilnika, kot jo ima CPU. Na sliki 3.1 lahko vidimo primerjavo naraščanja zmogljivosti CPU in GPU. Dandanes grafične procesne enote najdemo v skoraj vseh računalnikih. Izjema niso niti superračunalniki, ki jih imajo po več tisoč.

Glavna lastnost, zaradi katere lahko grafične procesne enote upodabljajo slike hitreje kot centralne procesne enote, je njihova visoko paralelna arhitektura, ki jim omogoča sočasno izvajanje številnih izračunov. Zaradi svoje učinkovitosti se grafične procesne enote čedalje bolj uporabljajo tudi za



Slika 3.1: Primerjava števila operacij v plavajoči vejici na sekundo med CPU in GPU. Povzeto po [25].

računsko zahtevne probleme, ki niso neposredno povezani z računalniško grafiko. Govorimo o splošno-namenskem računanju na grafičnih procesnih enotah (angl. *General-purpose computing on graphics processing units, GPGPU*). Razlog za širjenje GPGPU vidimo predvsem v vsesplošni razpoložljivosti in cenovni dostopnosti grafičnih kartic. Pripomore tudi dejstvo, da se zaradi večjedrnih procesorjev vse bolj uveljavlja paralelno programiranje. Ravno pri paralelnosti pa se zelo dobro izkažejo grafične procesne enote, saj so zmožne hkrati izvajati več deset tisoč vzporednih niti.

Grafična procesna enota je zaradi svojih grafičnih korenin najbolj primerna za reševanje problemov z naslednjimi karakteristikami [26]:

1. **Obsežna računska zahtevnost**
   Upodabljanje v realnem času zahteva izris več milijonov točk na sekundo, vsaka točka pa zahteva več sto operacij. GPU ima ogromne računske zmogljivosti, da zadovolji potrebe realno-časovnih aplikacij.

2. **Visoka stopnja podatkovnega paralelizma**
   Arhitektura grafičnega cevovoda je zaradi operacij nad točkami in fragmenti primerna za izvajanje paralelnih programov, kar lahko izkoristimo pri mnogih drugih računskih problemih.

3. **Prepustnost je bolj pomembna kot zakasnitev**
   Na CPU je poudarek na čim hitrejšem izvajanju ene niti, za kar potrebujemo predpomnilnike za zmanjšanje zakasnitev in kompleksno kontrolno logiko. Na GPU pa se zakasnitve skrije s sočasnim, sicer počasnejšim, izvajanjem večjega števila niti. Če mora ena nit čakati na podatek iz pomnilnika, se medtem izvaja druga nit. Računski problemi s poudarkom na prepustnosti niso izključno značilni za računalniško grafiko, ampak jih najdemo tudi v mnogih drugih domenah.

Probleme, ki ustrezajo zgornjim karakteristikam, lahko najdemo na najrazličnejših področjih, od fizike, astrofizike in kemije, pa do medicine, biologije in genetike, na primer:

- simulacija fizikalnih modelov,
- problemi N teles,
- dinamika tekočin,
- modeliranje vremena in obnašanja okolje,
- procesiranje slik,
- iskanje in urejanje,
- določanje zgradbe proteinov in
- modeliranje celic.

3.1 CUDA

CUDA, *Compute Unified Device Architecture*, je strojna in programska platforma za vzporedno računanje na grafičnih procesnih enotah. Razvijalcem ponuja neposreden dostop do nabora ukazov računskih elementov na GPU in do njihovega pomnilnika. Platformo so razvili pri podjetju NVIDIA. Ker

gre za zaprto, pa čeprav brezplačno platformo, je ta na voljo le na grafičnih procesnih enotah podjetja NVIDIA. Grafične procesne enote z arhitekturo CUDA so na voljo od novembra 2006, ko je izšla grafična kartica NVIDIA *GeForce 8800 GTX*, prva kartica z mikroarhitekturo NVIDIA Tesla.

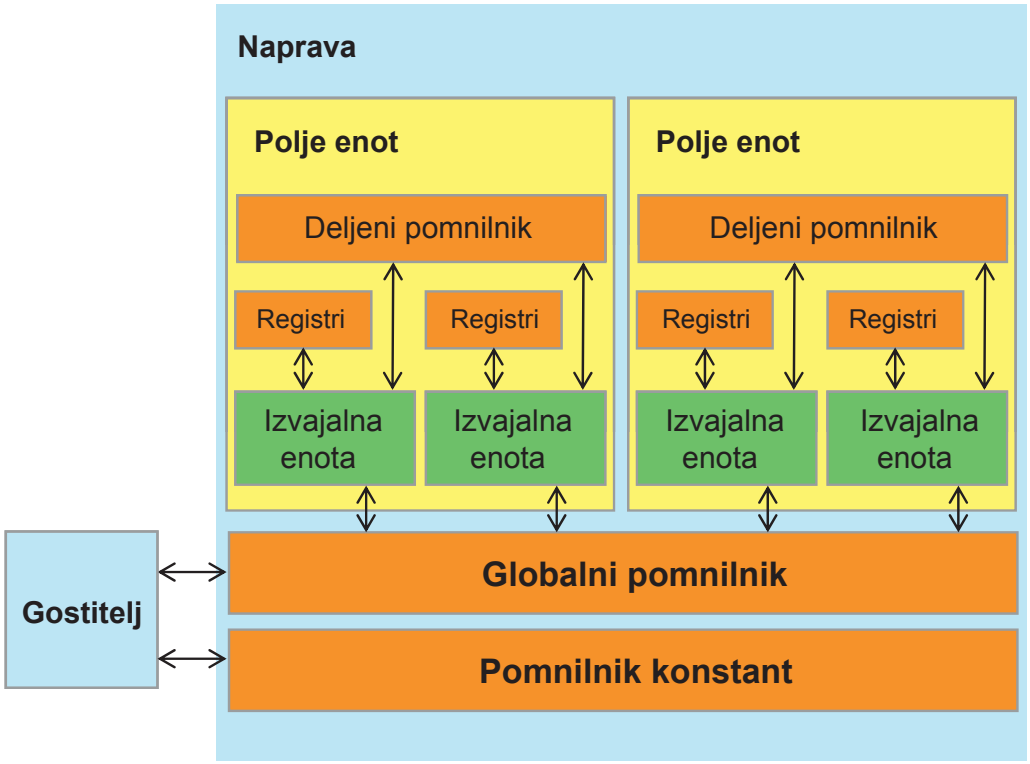
Programe za platformo CUDA lahko pišemo v razširitvah programskih jezikov C, C++ in Fortran, imenovanih CUDA C/C++ in CUDA Fortran. Seveda so na voljo tudi knjižnice, ki nam omogočajo pisanje programov CUDA v drugih popularnih programskih jezikih, kot so Python, C#, Java in Matlab. Obstajajo pa tudi visoko-nivojski vmesniki, ki programiranje ovijejo v dodaten nivo abstrakcije in s tem programerju olajšajo delo. Primer takšnega vmesnika je programski jezik *hiCUDA* [27].

3.1.1 Arhitektura

Pred arhitekturo CUDA so bile izvajalne enote v GPU razdeljene na senčilnike vozlišč in senčilnike slikovnih točk. V nekaterih aplikacijah se je lahko zgodilo, da so bile ene enote preobremenjene, druge pa neizkoriščene. Z arhitekturo CUDA so to posplošili in sedaj lahko vsaka izvajalna enota na GPU izvaja vse operacije. Razširili so nabor ukazov in ga, namesto prejšnje specializacije zgolj za grafiko, prilagodili splošnemu računanju. Aritmetično-logične enote so načrtovali tako, da so skladne s standardom IEEE 754 za računanje v plavajoči vejici. Poleg tega so posameznim izvajalnim enotam dovolili naključen dostop do pomnilnika, torej lahko pišejo v ali berejo iz poljubne pomnilniške besede. Vse našteto so dodali z namenom, da bi ustvarili grafično procesno enoto, ki bi se odlikovala tudi pri splošnih računskih problemih in ne le pri tradicionalnih grafičnih nalogah [28].

Arhitektura CUDA (slika 3.2) je zgrajena okrog polja enot, imenovanih *Streaming Multiprocessors (SM)*. Vsako polje enot je sestavljeno iz več izvajalnih enot, imenovanih *Streaming Processors (SP)*, v novejših GPU-jih tudi *CUDA Core*. Polje enot je pravzaprav samostojen procesor, ki skrbi za sočasno izvajanje več sto niti. To mu omogoča arhitektura SIMT (*Single-Instruction, Multiple-Thread*) [29], kar pomeni, da več niti hkrati izvaja isti

ukaz. Vsaka nit se izvaja v eni izvajalni enoti.



Slika 3.2: Arhitektura in pomnilniški model CUDA. Povzeto po [30].

Pomnilniški model (slika 3.2) je zasnovan na različnih tipih pomnilnika.

1. **Registri** (angl. *Registers*)
   So najhitrejši tip pomnilnika, ki je na voljo vsaki niti. Čas dostopa znaša le eno urino periodo. Zavedati pa se moramo, da jih je zelo malo.

2. **Lokalni pomnilnik** (angl. *Local memory*)
   Ko zmanjka prostora v registrih, se podatki shranjujejo v lokalni pomnilnik. Podobno kot pri registrih lahko vsaka nit dostopa le do svojega dela pomnilnika. Ker je lokalni pomnilnik fizično del globalnega, so zakasnitve pri dostopu relativno velike.

3. **Deljeni pomnilnik** (angl. *Shared memory*)
   Fizično se nahaja znotraj polja enot, zato lahko do njega dostopajo le niti v istem bloku. Dostop do deljenega pomnilnika je precej hitrejši od dostopa do globalnega pomnilnika in le malce počasnejši od dostopov do registrov. Čas dostopa znaša med 10 in 20 urinih period. Uporabljamo ga lahko za komunikacijo in deljenje podatkov med nitmi v istem bloku.

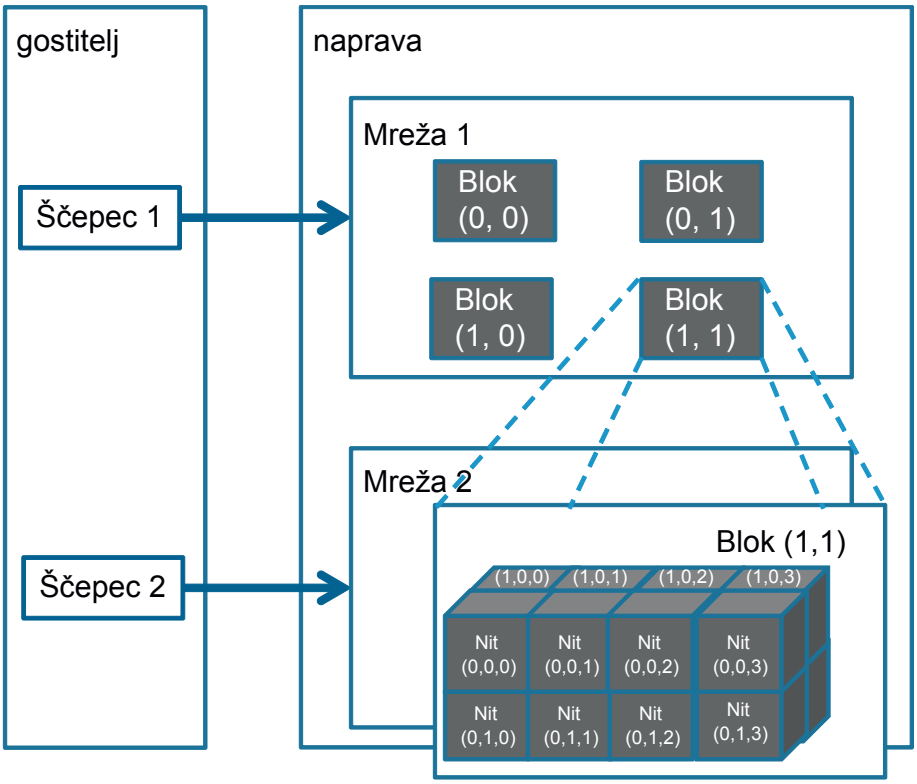
4. **Globalni pomnilnik** (angl. *Global memory*)
   Je največji izmed naštetih vrst pomnilnikov, vendar je tudi dostop do njega najpočasnejši. Čas dostopa znaša med 400 in 800 urinih period. Implementiran je v tehnologiji DRAM in ni na istem čipu kot GPU. Do njega lahko dostopajo vse niti, ne glede na blok. Ker je glavni namen globalnega pomnilnika komunikacija in prenos podatkov med gostiteljem in napravo, je dostopen tudi iz CPU.

5. **Pomnilnika konstant in tekstur** (angl. *Global and Texture memory*)
   Oba sta specializirana bralna pomnilnika in sta ostanka grafičnih korenin GPU. Implementirana sta v tehnologiji DRAM in sta predpomnjena.

3.1.2 Izvajalni model

Včasih je bilo potrebno računske probleme prevesti v grafične probleme, če smo jih želeli reševati na GPU. Uporabljati smo morali grafično orientirane senčilne jezike kot sta OpenGL-ov GLSL in Microsoftov HLSL. S prihodom platforme CUDA to ni več potrebno.

Program CUDA sestoji iz dveh delov. Prvi del je serijski program, ki se v eni (ali več) niti izvaja na gostitelju (CPU). Drugi del pa je eden ali več ščepcev (angl. *kernels*). Ščepec je program, ki se izvaja na napravi GPU. Ko ščepec v sekvenčni kodi zaženemo, ga na napravi začne izvajati več vzporednih niti. Tu lahko izkoristimo podatkovni paralelizem in ščepec sprogramiramo tako, da ga vsaka nit izvaja nad različnimi podatki. Ko se izvajanje ščepca konča, se nadaljuje izvajanje sekvenčne kode na gostitelju.

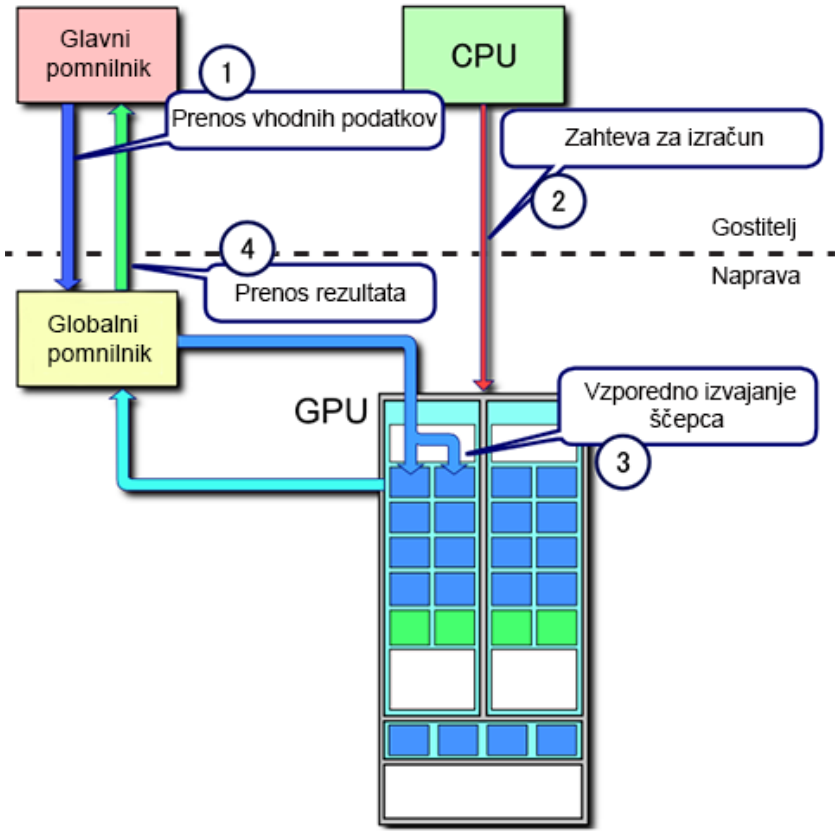


Slika 3.3: Logična organizacija niti. Povzeto po [30].

Niti so logično organizirane v bloke (angl. *blocks*), bloki pa v mrežo (angl. *grid*)(slika 3.3). Bloki in mreže so lahko eno-, dvo- ali pa tri-dimenzionalni. Vsaka nit je tako dostopna preko eno-, dvo- ali tri-dimenzionalnih indeksov. Dimenzionalnost in velikost blokov in mrež določi programer. Običajno jih prilagodi strojni opremi in dimenzijam problema, ki ga rešuje. Če na primer množimo dvodimenzionalne matrike, je najbolj enostavno, da so tudi niti organizirane v dvodimenzionalne bloke.

Celoten blok niti je dodeljen za izvajanje enemu polju enot. Na enem polju enot se lahko hkrati izvaja več blokov niti. V kolikor polje enot nima dovolj resursov za vzporedno izvajanje blokov, se ti izvajajo v poljubnem vrstnem redu. Niti znotraj enega bloka lahko med seboj komunicirajo preko deljenega pomnilnika, lahko pa jih med seboj tudi sinhroniziramo. Vse niti lahko med seboj sinhroniziramo zgolj na gostitelju.

Izvajanje tipičnega programa CUDA poteka v štirih korakih (slika 3.4). Najprej moramo iz pomnilnika gostitelja v pomnilnik naprave prenesti vhodne podatke. Nato gostitelj izda zahtevo za zagon ščepca. Ta se začne izvajati v več sočasnih nitih na napravi in pri tem dostopa in piše v pomnilnik naprave. Sledi še prenos rezultatov nazaj v pomnilnik gostitelja.



Slika 3.4: Potek izvajanja tipičnega programa CUDA. Povzeto po [31].

Poglavje 4

Implementacija algoritma OMES

Glavni cilj diplomske naloge je paralelizacija algoritma OMES in implementacija na grafični procesni enoti. Uspešnost paralelizacije smo preverili v primerjavi s sekvenčno različico algoritma, ki se izvaja na centralni procesni enoti.

Kodo, ki se izvaja na CPU smo napisali v programskem jeziku C++, kodo, ki se izvaja na GPU pa v programskem jeziku CUDA C. C++ je visokonivojski objektno-orientiran programski jezik, ki pa nam hkrati omogoča tudi nizko-nivojsko upravljanje s pomnilnikom in druge optimizacije. CUDA C je razširitev programskega jezika C z novimi ključnimi besedami in programskimi vmesniki, ki nam omogoča programiranje na grafičnih procesnih enotah. Sekvenčno implementacijo smo razvijali v integriranem razvojnem okolju Microsoft Visual Studio 2013, implementacijo na GPU pa v razvojnem okolju NVIDIA Nsight Eclipse Edition.

V poglavju 4.1 smo podrobneje opisali našo sekvenčno implementacijo in se bolj osredotočili na sam postopek izračuna vrednosti p (enačba 2.4) in končnega rezultata, v poglavju 4.2 pa smo opisali težave, na katere smo naleteli ob implementaciji na GPU, in kako smo jih rešili.

4.1 Sekvenčna implementacija

4.1.1 Vhodni podatki

Program potrebuje štiri vhodne podatke. Prvi je parameter N, ki nam pove kolikokrat bomo premešali vsako mesto v eni izmed poravnav več zaporedij (poglavje 2.2.2). Naslednji je parameter pTh, ki predstavlja zgornjo mejo vrednosti p, ki nas bolj podrobno zanimajo. Uporabimo ga pri generiranju karte proteinskih stikov (poglavje 2.2.3) in poskrbi da imajo majhne vrednosti p na voljo več odtenkov sivin.

Za delovanje naš program potrebuje še dve poravnavi več zaporedij, vsako v svoji datoteki. Datoteki morata biti strukturirani tako, da vsaka vrstica predstavlja eno zaporedje. Vsaka vrstica mora biti sestavljena iz identifikatorja zaporedja in zaporedja samega. Vsako zaporedje mora biti sestavljeno iz niza znakov (velikih tiskanih črk), kjer vsak znak predstavlja en aminokislinski ostanek. Znak - pomeni vrzel v zaporedju, ki je nastala pri poravnavanju zaporedij. Vsa zaporedja morajo biti enake dolžine. Prazne vrstice program preskoči. Primer takšne datoteke lahko vidimo na sliki 2.2. Ker bomo pri delu s poravnavami več zaporedij operirali z matrikami, bomo v tem poglavju namesto o mestih v poravnavi govorili o stolpcih.

Naš program ob zagonu najprej prebere obe poravnavi več zaporedij iz datotek. V ta namen smo napisali razred `MSA` (Izsek kode 4.1), ki hrani vse podatke o prebranih poravnavah in vsebuje metode za branje, mešanje in druge operacije povezane z njimi. Dejansko branje iz datoteke in razčlenjevanje poravnave se zgodi v metodi `void MSA::ParseFromFile(std::string filename)`. Tu zaporedja preberemo in jih shranimo v pomnilnik v polje `char *m_Data`.

Pri tem moramo upoštevati da je pomnilnik linearen, vsebina datoteke pa je dvodimenzionalna matrika (poravnava več zaporedij je zapisana v več vrsticah in stolpcih). Poznamo več načinov predstavitve večdimenzionalnih matrik v linearnem pomnilniku. Najpogosteje uporabljana načina sta ureditev po vrsticah (angl. *row-major order*) in ureditev po stolpcih (angl. *column-major order*). Pri prvem načinu si v pomnilniku eden za drugim sledijo ele-

Izsek kode 4.1: Deklaracija razreda `MSA`. (datoteka `CMA.h`)

```
1  class MSA
2  {
3    private:
4      char *m_Data = nullptr; // MSA (column major matrix)
5      std::map<char, float> *m_FreqRelative = nullptr;
6      bool *m_SelectedPositions = nullptr;
7
8    public:
9      size_t NumOfSequences = 0;  // height
10     size_t SequenceLength = 0;  // width
11
12     MSA(char* filename);
13     ~MSA();
14     void ParseFromFile(std::string filename);
15     void CalculateFrequencies();
16     char Get(int row, int column);
17     std::map<char, float>& GetRelativeFrequency(int index);
18     bool IsColumnSelected(int column);
19     void Shuffle();
20 };
```

menti vrstic matrike, pri drugem pa elementi stolpcev. Za naš problem je bolj primerna ureditev po stolpcih, saj pri računanju pričakovanih in opazovanih frekvenc najpogosteje po vrsti dostopamo do vseh elementov v posameznem stolpcu. Posledica te prostorske lokalnosti je manj zgrešitev v predpomnilniku in hitrejše delovanje programa [32].

Spremenljivko `m_Data` bi sicer lahko deklarirali kot dvodimenzionalno polje (`char m_Data[][]`), vendar v programskem jeziku C++ to pomeni, da bi se za predstavitev v pomnilniku avtomatsko uporabljala ureditev po vrsticah. Zato smo spremenljivko `m_Data` deklarirali kot enodimenzionalno polje (oziroma kot kazalec na enodimenzionalno polje). Do elementa poravnave v i-tem zaporedju (vrstici originalne datoteke) in j-tem stolpcu lahko sedaj dostopamo z `m_Data[index]`, kjer je `index` enak $i+j\cdot$*število zaporedij*. Lahko pa uporabimo tudi metodo `char MSA::Get(int row, int column)`, ki `index` izračuna namesto nas.

4.1.2 Izračun relativnih frekvenc

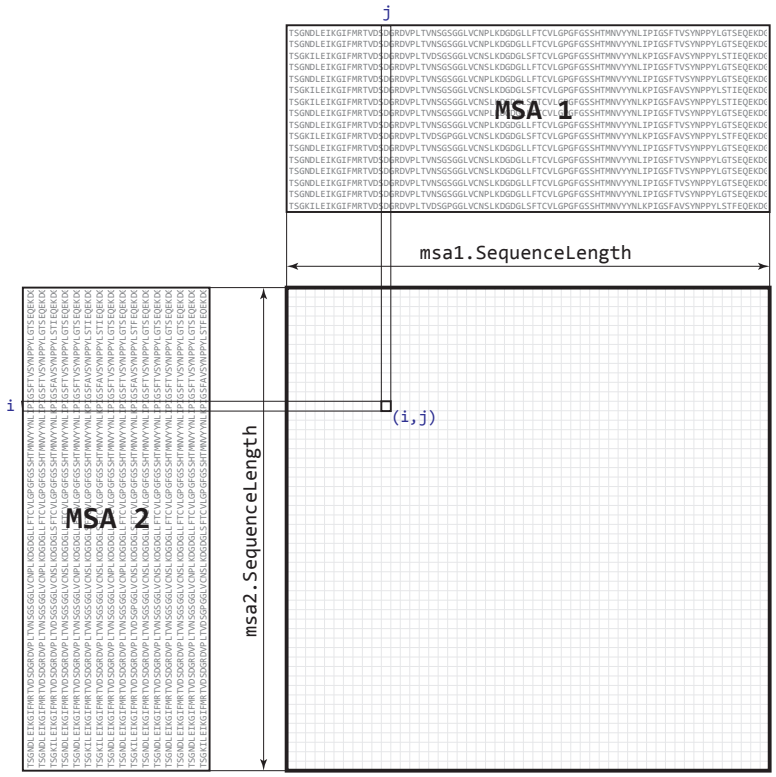
Relativne frekvence pojavitev posameznega aminokislinskega ostanka v vsakem stolpcu poravnave več zaporedij izračunamo tako, da preštejemo kolikokrat se določen aminokislinski ostanek v stolpcu pojavi in to število delimo s številom vseh aminokislinskih ostankov v stolpcu. Ker relativne frekvence potrebujemo v naslednjih korakih algoritma, jih moramo shraniti. Za to je najbolj primerna podatkovna struktura slovar (angl. *dictionary*, tudi *map*). Slovar je pravzaprav množica parov ključ-vrednost. Do posamezne vrednosti lahko dostopamo preko njenega ključa. V našem primeru bo ključ črka, ki predstavlja aminokislinski ostanek, vrednost pa relativna frekvenca tega aminokislinskega ostanka v določenem stolpcu. V programskem jeziku C++ je podatkovna struktura slovar implementirana z razredom `std::map` iz standardne knjižnice. Ker si moramo relativne frekvence zapomniti za vsak stolpec posebej, potrebujemo za vsak stolpec svoj slovar. Relativne frekvence izračunamo v metodi `void CalculateFrequencies()` in jih shranimo v polje slovarjev `std::map<char, float> *m_FreqRelative` v razredu `MSA` (izsek kode 4.1).

Zanimajo nas le stolpci, ki vsebujejo vsaj dva različna aminokislinska ostanka, izključujoč vrzeli. V nasprotnem primeru bo stolpec po mešanju vedno enak, zato ga lahko pri računanju preskočimo. Stolpce, ki nas zanimajo, označimo z vrednostjo `true` na ustreznem mestu v polju `m_SelectedPositions` v razredu `MSA`.

4.1.3 Glavni del algoritma

Za vsako možno kombinacijo stolpcev v obeh poravanavh več zaporedij moramo primerjati pričakovane in opazovane frekvence pojavitev aminokislinskih ostankov. Vse možne kombinacije stolpcev najlažje predstavimo z matriko. Ta matrika mora imeti toliko stolpcev kot je dolžina zaporedij v prvi poravnavi in toliko vrstic kot je dolžina zaporedij v drugi poravnavi. Primer takšne matrike vidimo na sliki 4.1. Vsak element matrike predstavlja eno možno kombinacijo stolpcev obeh poravnav. Element v i-ti vrstici in j-tem

stolpcu matrike ustreza kombinaciji j-tega stolpca v prvi poravnavi in i-tega stolpca v drugi poravnavi.



Slika 4.1: Primer matrike vseh možnih kombinacij stolpcev MSA.

V algoritmu potrebujemo več takih matrik. Matrika `map<string, float> *matrixExpectedFreqs` vsebuje pričakovane frekvence za vsako možno kombinacijo stolpcev. Za zapis pričakovanih frekvenc ene kombinacije stolpcev smo ponovno uporabili slovar. Tokrat bodo ključi možni pari aminokislinskih ostankov, ki se lahko pojavijo v določeni kombinaciji stolpcev, vrednosti pa pričakovane frekvence pojavitev teh parov. Potrebujemo tudi matriko `float *matrixChiSq0`, kamor shranjujemo vrednosti χ_0^2, izračunane pred prvim mešanjem. Zadnja matrika, ki jo potrebujemo, je matrika `int * matrixChiSqCounts`. V tej matriki bomo po enačbi 2.5 za vsako možno kombinacijo stolpcev posebej šteli, kolikokrat je vrednost χ_r^2, izračunana po vsakem mešanju, večja od vrednosti χ_0^2.

Izsek kode 4.2: Glavni del sekvenčnega algoritma. (datoteka `Cma.cpp`)

```
int matrixLength = (int) (msa1.SequenceLength * msa2.SequenceLength);
float *matrixChiSq0 = new float[matrixLength];
int *matrixChiSqCounts = new int[matrixLength];
for( int i = 0; i < matrixLength; i++ )
  matrixChiSqCounts[i] = 0;
map<string, float> *matrixExpectedFrequencies = new map<string, float>[
    matrixLength];

for( int n = 0; n < N; n++ )
{
  int index = 0;
  for( int row = 0; row < msa2.SequenceLength; row++ )
  {
    for( int column = 0; column < msa1.SequenceLength; column++, index++ )
    {
      // ignore columns that contain less than 2 aminoacids
      if( !msa1.IsColumnSelected(column)  !msa2.IsColumnSelected(row) )
        continue;

      if( n == 0 )
        CalculateExpectedFrequencies( (float) msa1.NumOfSequences,
          msa1.GetRelativeFrequency( column ),
          msa2.GetRelativeFrequency( row ),
          matrixExpectedFrequencies[index] );

      map<string, int> observed;
      map<string, float>& expected = matrixExpectedFrequencies[index];
      CountObservedFrequencies( msa1, column, msa2, row, observed );

      float chiSq = CalculateChiSq( expected, observed );

      float chiSqUnshuffled;
      if( n == 0 )
        matrixChiSq0[index] = chiSqUnshuffled = chiSq;
      else
        chiSqUnshuffled = matrixChiSq0[index];

      if( chiSq >= chiSqUnshuffled )
        matrixChiSqCounts[index]++;
    }
  }
  msa1.Shuffle();
}
```

Implementacijo glavnega dela algoritma lahko vidimo na izseku kode 4.2. Z zunanjo zanko N-krat ponovimo računanje vrednosti χ_r^2 in mešanje stolpcev. Z notranjima zankama se sprehodimo čez vse možne kombinacije stolpcev. Pri tem spremenljivka `row` predstavlja indeks vrstice v matriki vseh možnih kombinaciji stolpcev in hkrati tudi indeks trenutnega stolpca v drugi poravnavi več zaporedij (slika 4.1). Spremenljivka `column` pa predstavlja indeks stolpca v matriki vseh možnih kombinaciji stolpcev in indeks trenutnega stolpca v prvi poravnavi več zaporedij.

Kot smo že omenili, nas stolpci, ki ne vsebujejo vsaj dveh različnih aminokislinskih ostankov, ne zanimajo, zato jih pri računanju preskočimo (vrstice 15-17).

Za vsako možno kombinacijo stolpcev moramo najprej izračunati pričakovane frekvence pojavitev aminokislinskih ostankov N_{EX}. To storimo v funkciji `CalculateExpectedFrequencies`. Vse možne kombinacije dobimo tako, da izračunamo kartezični produkt med množicama aminokislinskih ostankov, ki se pojavijo v stolpcih trenutne kombinacije. Za vsak par izračunamo še njegovo pričakovano frekvenco po enačbi 2.2. Pri tem uporabimo že izračunane relativne frekvence (poglavje 4.1.2). Rezultate shranimo v slovar in ga zapišemo na ustrezno mesto v matriki `matrixExpectedFreqs`. Ker se pričakovane frekvence po mešanju ne bodo spreminjale, je dovolj, če jih izračunamo samo v prvem obhodu zunanje zanke.

stolpec

i	j
A	C
A	C
B	C
B	C
B	D

relativne frekvence

i	j
A: 0.4	C: 0,8
B: 0.6	D: 0,2

pričakovane in opazovane frekvence

	N_{EX}	N_{OBS}
AC	1,6	2
AD	0,4	0
BC	2,4	2
BD	0,6	1

Tabela 4.1: Primer izračuna pričakovanih in opazovanih frekvenc za kombinacijo stolpcev i in j.

Za vsako možno kombinacijo stolpcev moramo izračunati tudi opazovane frekvence N_{OBS}. To storimo v funkciji `CountObservedFrequencies`, kjer preštejemo kolikokrat se posamezen par aminokislinskih ostankov dejansko pojavi v trenutni kombinaciji stolpcev. Primer izračuna pričakovanih in opazovanih frekvenc lahko vidimo v tabeli 4.1.

V naslednjem koraku kličemo funkcijo `CalculateChiSq`. Na podlagi pravkar izračunanih opazovanih in pričakovanih frekvenc izračunamo vrednost χ_r^2 za trenutno kombinacijo stolpcev po enačbi 2.3. V kolikor gre za prvi obhod zunanje zanke, torej še pred prvim mešanjem, izračunano vrednost χ_0^2 shranimo v matriko `matrixChiSq0` (vrstica 35).

V zadnjem koraku algoritma še primerjamo vrednosti χ_r^2 z vrednostmi χ_0^2. Če je vrednost χ_r^2 večja od vrednosti χ_0^2, povečamo vrednost na ustreznem mestu v matriki `matrixChiSqCounts` za 1 (enačba 2.5).

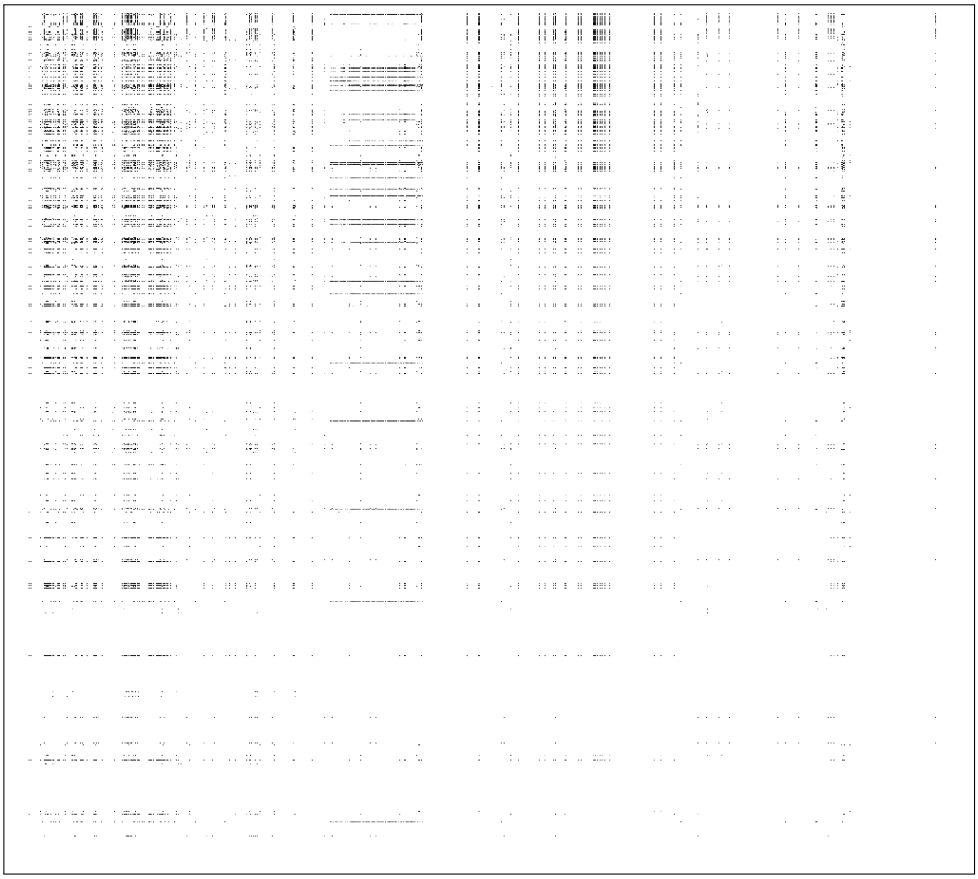
Ko smo to storili za vse možne kombinacije stolpcev, s klicem funkcije `msa1.Shuffle()` premešamo prvo poravnavo več zaporedij. Natančneje, v vsakem stolpcu prve poravnave naključno premešamo aminokislinske ostanke. Za mešanje smo uporabili funkcijo `std::random_shuffle(char* first,char* last)` iz standardne knjižnice C++. Ta zagotavlja, da je vsaka možna permutacija enako verjetna.

4.1.4 Izhodni podatki

Po zaključku zunanje zanke izračunamo še dejanske vrednosti p. Dobimo jih tako, da vrednosti v matriki `matrixChiSqCounts` delimo s številom mešanj (enačba 2.4). Na mesta, za katera vrednosti p nismo računali, vstavimo vrednost 1. Če bi tudi za ta mesta računali vrednost p, bi namreč vedno dobili rezultat 1, ker bi bile vrednosti χ_r^2 po vsakem mešanju vedno enake vrednostim χ_0^2. Tako smo pri računanju prihranili kar nekaj časa.

Dobljeno matriko vrednosti p izpišemo v datoteko in jo prikažemo še kot karto proteinskih stikov. Primer takšne karte lahko vidimo na sliki 4.2. Zaradi enostavnosti smo za format slike izbrali format PGM (angl. *Portable Graymap Format*). V tem formatu je vsaka točka zapisana s številom med 0

(črna barva) in 255 (bela barva). Ker nas bolj podrobno zanimajo samo zelo majhne vrednosti p, pobarvamo vrednosti p, ki so večje kot pTh belo, vse ostale pa premo sorazmerno preslikamo na interval $[0, 255]$. Tako zagotovimo da imajo majhne vrednosti p na voljo večji razpon odtenkov sive barve.



Slika 4.2: Primer izhodne slike pri $N = 1000$ in $pTh = 0,025$.

4.2 Posebnosti implementacije na GPU

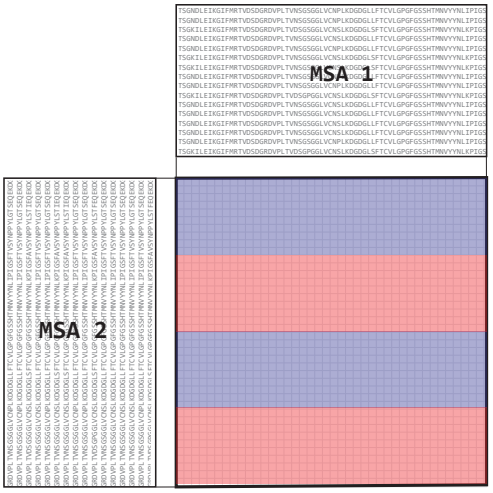
4.2.1 Razdelitev dela

Pri programiranju na grafični procesni enoti imamo na voljo veliko število vzporednih niti. Našo sekvenčno implementacijo algoritma moramo torej

prilagoditi tako, da bo računanje lahko izvajalo več niti vzporedno. To najbolj enostavno storimo tako, da vsaka nit izračuna vrednost p ene od možnih kombinacij stolpcev. Tako so niti med seboj povsem neodvisne in ne potrebujejo medsebojne sinhronizacije. Za izračun vrednosti p nit potrebuje le podatke iz svojega para stolpcev, za katerega računa vrednost p.

Za predstavitev vseh možnih kombinacij stolpcev smo v poglavju 4.1.3 uporabili dvodimenzionalno matriko. Zato je smiselno, da tudi niti organiziramo v dvodimenzionalne bloke in mreže. Mreža niti bo torej enako velika kot matrika vseh možnih kombinacij stolpcev s slike 4.1. Tako lahko vsako nit identificiramo z dvodimenzionalnim indeksom, ki hkrati ustreza tudi indeksu elementa v matriki vseh možnih kombinacij, ki ga ta nit računa.

Če hočemo vzporedno zagnati vse te niti, moramo prej seveda v pomnilnik grafične procesne enote prenesti vse potrebne vhodne podatke. Poleg tega moramo tam rezervirati prostor, ki ga potrebujemo za hranjenje matrik pričakovanih frekvenc, opazovanih frekvenc in končnih rezultatov. Ker so vse te strukture zelo velike, se lahko zgodi, da nam pri poravnavah z zelo dolgimi zaporedji zmanjka pomnilnika. Ker smo hoteli implementirati računanje tudi za poljubno velike poravnave več zaporedij, smo matriko vseh možnih kombinacij stolpcev razsekali na kose (slika 4.3). Naenkrat smo zagnali le niti, ki računajo vrednosti p kombinacij stolpcev v trenutnem kosu. Tako na grafični procesni enoti potrebujemo le toliko pomnilnika, da vanj shranimo podatke, potrebne za računanje enega kosa. Preden lahko začnemo računati na naslednjem



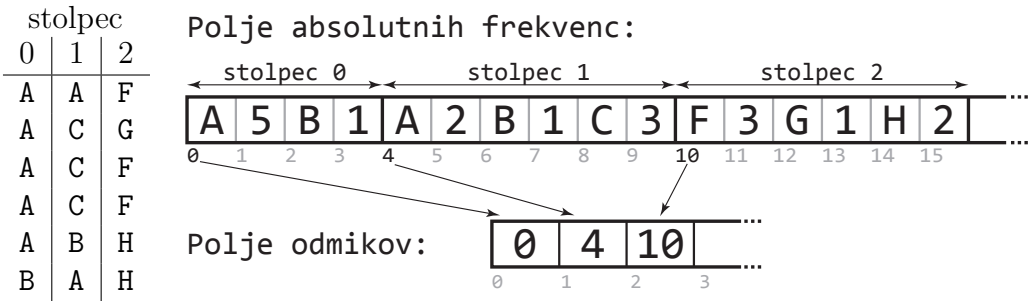
Slika 4.3: Razdelitev matrike vseh možnih kombinacij na kose.

kosu, pa moramo rezultate prejšnjega kosa prenesti nazaj na gostitelja in na grafično procesno enoto prenesti potrebne podatke za naslednji kos.

4.2.2 Slovarji na GPU

Pri sekvenčni implementaciji smo se za operacije z relativnimi in pričakovani frekvencami zanašali na podatkovno strukturo slovar, natančneje na `std::map` iz standardne C++ knjižnice. Ker na grafični procesni enoti takšne strukture niso na voljo, smo poiskali alternativno rešitev.

Za izračun relativnih frekvenc (poglavje 4.1.2) moramo najprej prešteti kolikokrat se pojavi vsak aminokislinski ostanek v vsakem stolpcu poravnave več zaporedij. To lahko najelegantneje storimo na centralni procesni enoti z uporabo podatkovne strukture slovar ob branju poravnav iz datoteke. Ker smo vse nadaljnje računanje prestavili na grafično procesno enoto, moramo tja prenesti absolutne frekvence pojavitev aminokislinskih ostankov.



Slika 4.4: Primer polja absolutnih frekvenc in pripadajočih odmikov.

Absolutne frekvence smo zapisali v enodimenzionalno polje, v formatu prikazanem na sliki 4.4. Vsakemu aminokislinskemu ostanku, ki se pojavi v stolpcu, sledi število njegovih pojavitev. Stolpci si v polju sledijo eden za drugim. Opazimo, da je v nekaterih stolpcih lahko več različnih aminokislinskih ostankov kot v drugih. Da vemo, kje v polju absolutnih frekvenc se začnejo podatki določenega stolpca, potrebujemo še polje odmikov (angl. *offset*, tudi *displacement*). Element z indeksom i v polju odmikov nam pove na katerem indeksu v polju absolutnih frekvenc se začnejo podatki o absolu-

tnih frekvencah stolpca i. Ker bomo pri nadaljnjem računanju do absolutnih frekvenc vedno dostopali po vrsti, je takšna implementacija slovarja povsem zadostujoča.

Drugače pa je s pričakovanimi in opazovanimi frekvencami. Pri računanju vrednosti χ^2 za neko kombinacijo stolpcev po enačbi 2.3 moramo po vrsti dostopati do vseh opazovanih frekvenc parov aminokislinskih ostankov, ki se na tej kombinaciji stolpcev pojavijo. Za vsak tak par aminokislinskih ostankov moramo poiskati tudi njegovo pričakovano število pojavitev. Če bi pričakovane frekvence shranili v enodimenzionalno polje na enak način kot absolutne frekvence, bi iskanje pričakovane frekvence za nek par imelo časovno zahtevnost $O(n)$. Ker pari v tem polju niso urejeni, bi morali vedno po vrsti dostopati do vseh parov, dokler ne bi našli iskanega. Za primerjavo, iskanje elementa v `std::map` ima časovno zahtevnost $O(\log n)$.

Ta problem smo zaobšli tako, da smo prilagodili format zapisa pričakovanih in opazovanih frekvenc v enodimenzionalno polje. Do sedaj smo aminokislinske ostanke označevali z znaki, enakimi kot so v datotekah iz katerih preberemo poravnave več zaporedij. Za same izračune je pomembno zgolj razlikovanje med različnimi aminokislinski ostanki v posameznem stolpcu. V vsakem stolpcu lahko torej vsak različen znak preslikamo v zaporedno številko njegove pojavitve. Izjema je samo znak `-` za vrzel, ki ga ne spreminjamo. Primer takšne preslikave lahko vidimo v tabeli 4.2.

originalni stolpec

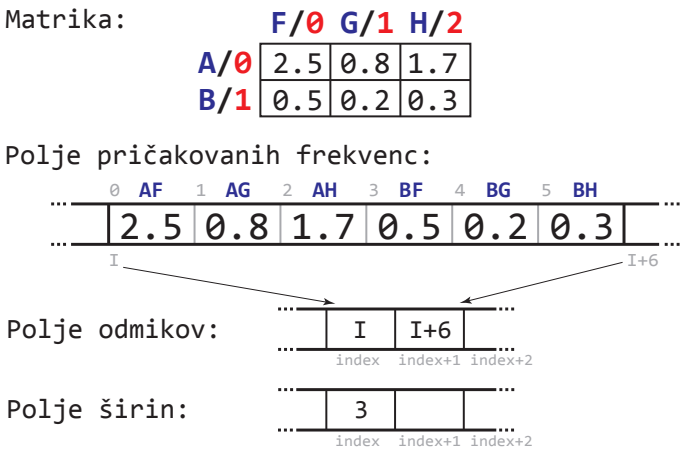
i	j
A	F
A	G
A	F
A	F
A	H
B	H

stolpec po preslikavi

i	j
0	0
0	1
0	0
0	0
0	2
1	2

Tabela 4.2: Primer preslikave znakov (aminokislinskih ostankov) v števila.

Ta preslikava nam omogoča da lahko pričakovane frekvence zapišemo v kompaktno dvodimenzionalno matriko, kot jo vidimo na sliki 4.5. Na tej



Slika 4.5: Primer polja pričakovanih frekvenc in pripadajočih odmikov. Uporabljena sta stolpca i in j iz tabele 4.2.

sliki smo za primer uporabili stolpca i in j iz tabele 4.2. Zaradi preslikave nam v matriko ni potrebno zapisovati znakov za aminokislinske ostanke, kot smo to storili pri absolutnih frekvencah. Katero število pripada kateremu paru aminokislinskih ostankov sedaj določajo preslikana števila. Pričakovano število pojavitev para aminokislinskih ostankov `B` in `H` (po preslikavi sta to števili $b = 1$ in $h = 2$) zapišemo v matriko v vrstico z indeksom b in stolpec z indeksom h. Kot je prikazano na sliki 4.5, lahko takšno matriko sploščimo v enodimenzionalno polje. Pričakovano število pojavitev para `BH` je sedaj zapisano v elementu polja z indeksom $h + b \cdot \textit{širina matrike}$. *Širina matrike* je enaka številu različnih aminokislinskih ostankov v stolpcu j. Ker je *širina matrike* različna za vsako kombinacijo stolpcev, jo moramo shraniti. Potrebujemo novo polje, v katerega na indeks *index* shranimo širino matrike pričakovanih frekvenc kombinacije stolpcev i in j. Ker tudi tu uporabljamo matriko kombinacij vseh stolpcev prikazano na sliki 4.1, velja $\textit{index} = j + i \cdot \textit{dolžina zaporedij v } MSA$. Da vemo, kje v polju pričakovanih frekvenc se začnejo podatki neke kombinacije stolpcev, ponovno potrebujemo še polje odmikov. Element z indeksom *index* v polju odmikov nam pove, na katerem indeksu v polju pričakovanih frekvenc se začnejo podatki o

pričakovanih frekvencah kombinacije stolpcev i in j.

Popolnoma enak način lahko uporabimo tudi za zapis opazovanih frekvenc. Pri tem lahko uporabimo kar ista polja odmikov in širin matrik, kot smo jih pri pričakovanih frekvencah. Širine matrik in odmiki so namreč enaki.

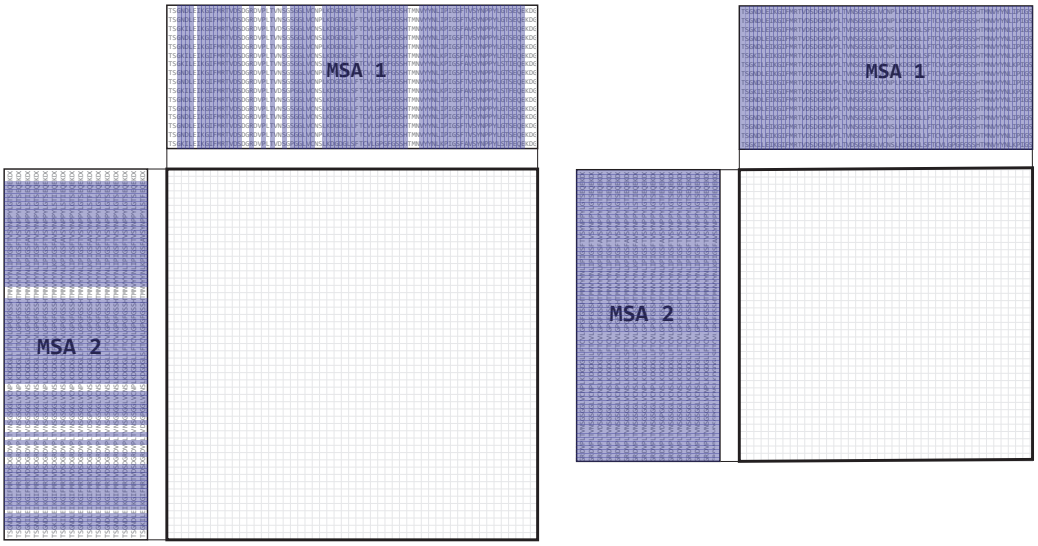
S takšnim formatom zapisa smo zagotovili, da je v polju pričakovanih frekvenc pričakovano število pojavitev nekega para aminokislinskih ostankov zapisano na isto ležečem mestu, kot je v polju opazovanih frekvenc zapisano število pojavitev istega para. Pri računanju vrednosti χ_r^2 nam sedaj pričakovanega števila ni več potrebno iskati.

4.2.3 Optimizacija neuporabljenih stolpcev

V poglavju 4.1.2 smo ugotovili, da nas stolpci poravnav več zaporedij, ki ne vsebujejo vsaj dveh različnih aminokislinskih ostankov, pravzaprav ne zanimajo. Takšne stolpce smo pri sekvenčni implementaciji označili v polju `*m_SelectedPositions` in jih pri računanju preskočili.

Če želimo računanje prestaviti na grafično procesno enoto, moramo tja prenesti vse podatke, ki jih pri računanju potrebujemo. Med drugim so to tudi podatki o obeh poravnavah več zaporedij. Ker smo s pomnilnikom na grafični procesni enoti bolj omejeni in sam čas prenosa podatkov ni zanemarljiv, se nam zdi nesmiselno, da bi tja prenašali tudi stolpce, ki jih pravzaprav ne bomo nikoli uporabili. Zato smo se odločili, da bomo pred prenašanjem odvečne stolpce odstranili. Posledično se, kot lahko vidimo na sliki 4.6, zmanjšajo tudi matrike, v katerih hranimo izračunane podatke o vseh možnih kombinacijah stolpcev. S tem še malce bolj zmanjšamo porabo pomnilnika na grafični procesni enoti in čas prenašanja podatkov tja in nazaj. Zmanjša se tudi število potrebnih niti. Brez te optimizacije bi namreč tudi za vsako nezanimivo kombinacijo morali zagnati ločeno nit. Šele pri računanju v ščepcu bi preverili, ali nas ta kombinacija sploh zanima.

Paziti pa moramo pri izpisu končnih rezultatov. Matrika izračunanih vrednosti p je sedaj manjša, kot bi bila, če stolpcev ne bi odstranjevali. Zato jo moramo pred izpisom povečati na ustrezno velikost. Na mesta, za katera



Slika 4.6: Na levi je primer matrike iz slike 4.1, kjer so z modro označeni stolpci, ki vsebujejo vsaj dva aminokislinska ostanka. Desno je primer iste matrike po odstranitvi odvečnih stolpcev.

nismo računali vrednosti p, moramo vstaviti ustrezne vrednosti, kot smo to storili v poglavju 4.1.4.

4.2.4 Mešanje stolpcev

Del algoritma OMES je tudi mešanje stolpcev. Če bi ohranili implementacijo mešanja iz sekvenčne implementacije (poglavje 4.1.3), bi morali rezultate mešanja vsakič prenašati na grafično procesno enoto. Ker se hočemo temu izogniti, moramo mešanje implementirati na grafični procesni enoti. Pri tem moramo celoten algoritem za mešanje implementirati sami, saj v ščepcih nimamo na voljo funkcije `random_shuffle` ali njej podobne.

Mešanje smo med niti razdelili tako, da vsaka nit premeša en stolpec. Če hočemo da bo mešanje naključno, moramo na grafični procesni enoti generirati naključna števila. Za to smo uporabili knjižnico cuRAND. Da bo vsako mešanje res naključno, moramo med mešanji shraniti stanje generatorja psevdonaključnih števil. Ker niti tečejo vzporedno, potrebuje vsaka

nit svoje stanje. Ker generatorji psevdonaključnih števil potrebujejo seme, moramo ta stanja tudi inicializirati. To storimo v posebnem ščepcu s klicem funkcije `curand_init` na začetku programa. Da pa ne bodo vse niti generirale istih psevdonaključnih števil, mora vsaka nit imeti različno seme. Zato skupnemu semenu dodamo še identifikator niti. Sedaj lahko v ščepcih generiramo psevdonaključna števila med 0 in 1 s klicem funkcije `curand_uniform(&state)`.

Izsek kode 4.3: Ščepec, s katerim premešamo MSA. (datoteka `kernels.cu`)

```
__global__ void kernel_shuffle(
  char *msa,  // MSA data
  int width,  // Sequence length
  int height, // Number of sequences
  curandState* states // array of PRNG states
){
  int columnIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if(columnIdx < width)
  {
    curandState state = states[columnIdx];  // get PRNG state

    // Fisher-Yates shuffle
    char *column = &msa[columnIdx*height];
    for(int i = height-1; i>0; i--)
    {
      int j = (int)(curand_uniform(&state) * i);
      char t = column[i];     //
      column[i] = column[j];  //  swap column[i] and column[j]
      column[j] = t;          //
    }
    states[columnIdx] = state;  // save PRNG state
  }
}
```

Eden najbolj znanih algoritmov za generiranje naključnih permutacij končne množice števil je algoritem Fisher-Yates. Za mešanje stolpcev smo v ščepcu na izseku kode 4.3 implementirali modernizirano verzijo tega algoritma. Sam algoritem je dokaj preprost. V polju, ki ga želimo premešati, izberemo naključen element in ga zamenjamo z zadnjim elementom v polju. Postopek nato ponavljamo, vendar v izbiro naključnega elementa ne vključujemo elementov, ki smo jih že prestavili na konec polja. Ko nam v polju zmanjka elementov, ki jih še nismo prestavili, smo končali. Rezultat je naključna permutacija originalnega polja.

Poglavje 5

Meritve in rezultati

5.1 Testno okolje in podatki

V okviru diplomske naloge smo algoritem OMES implementirali sekvenčno, za izvajanje na CPU, in paralelno, za izvajanje na platformi CUDA. Da bi ugotovili kakšne pohitritve smo dosegli z implementacijo na GPU, smo opravili meritve časov izvajanja programa.

Čase izvajanja za sekvenčni program smo merili na prenosnem računalniku s procesorjem Intel Core i5-3230M s frekvenco 2,60 GHz in 4 GB pomnilnika. Ker za poganjanje paralelnega programa potrebujemo grafično kartico podjetja NVIDIA, smo meritve izvajalnih časov paralelnega programa izvajali na delovni postaji z dvema šestjedrnima procesorjema Intel Xeon E5-2620, ki tečeta s frekvenco 1,20 MHz, 64 GB pomnilnika in dvema grafičnima karticama NVIDIA Tesla K20m. Podrobnosti o teh karticah lahko vidimo na sliki 5.1.

Na čas izvajanja programa najbolj vplivajo število korakov (število mešanj vsakega mesta) in velikost vhodnih poravnav več zaporedij. Te parametre smo spreminjali in za vsako kombinacijo opravili 3 meritve in izračunali povprečje. Pri velikosti poravnav smo spreminjali tako število zaporedij aminokislinskih ostankov, kot tudi dolžino le teh.

Poravnave več zaporedij, ki smo jih uporabljali kot vhod v naša programa,

```
Device 0: "Tesla K20m"
  CUDA Driver Version / Runtime Version          6.5 / 5.0
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 5120 MBytes (5368512512 bytes)
  (13) Multiprocessors x (192) CUDA Cores/MP:    2496 CUDA Cores
  GPU Clock rate:                                706 MHz (0.71 GHz)
  Memory Clock rate:                             2600 Mhz
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
```

Slika 5.1: Izsek izpisa programa CUDA Device Query, ki prikazuje lastnosti grafične kartice NVIDIA Tesla K20m.

smo pridobili iz projekta 1000 genomov [33]. Cilj tega projekta je vzpostavitev najbolj podrobnega kataloga človeške genetske variacije. V okviru tega projekta so bili sekvencirani genomi že več kot 1000 prostovoljcev.

5.2 Rezultati

Najprej smo izmerili, kako na čas izvajanja vpliva število korakov. En korak pomeni eno mešanje poravnave več zaporedij in izračun vrednosti χ^2 za vse kombinacije mest v poravnavi. Rezultate lahko vidimo v tabeli 5.1 in na sliki 5.2. Pohitritve smo izračunali po enačbi

$$S = \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU}}} \quad , \tag{5.1}$$

kjer je t_{CPU} celoten čas izvajanja sekvenčnega programa na CPU, t_{GPU} pa celoten čas izvajanja paralelnega programa na GPU. V t_{GPU} je vključena tudi vsa inicializacija, ki je potrebna za računanje na GPU. Ta vključuje alokacijo pomnilnika na GPU, prenašanje podatkov na in z GPU in ostale operacije povezane s tem.

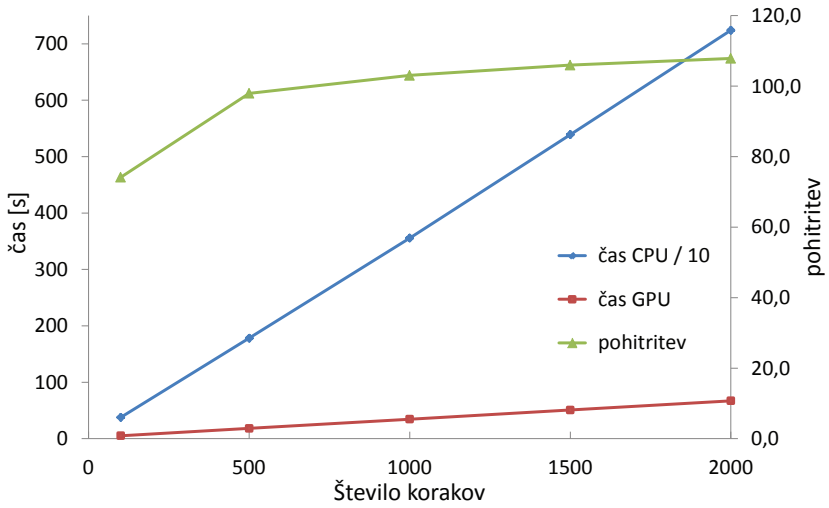
Kot smo pričakovali, je paralelna implementacija veliko hitrejša od sekvenčne implementacije. Dosegli smo namreč približno 100-kratno pohitritev. Če primerjamo še povprečne čase izvajanja enega koraka, ki na CPU

število korakov	t_{CPU} [s]	t_{GPU} [s]	S
100	378,1	5,1	74,1
500	1782,7	18,2	98,0
1000	3556,5	34,5	103,1
1500	5392,1	50,9	106,0
2000	7241,2	67,1	107,9

Tabela 5.1: Časi izvajanja in pohitritve pri različnem številu korakov in konstanti velikosti MSA (1000 zaporedij dolžine 1000 aminokislinskih ostankov).

traja 3,5 s, na GPU pa 0,03 s, smo dosegli celo 117-kratno pohitritev. Čas izvajanja enega koraka seveda ni odvisen od števila korakov.

Opazili smo, da pohitritev narašča s številom korakov. Razlog za to je inicializacija na GPU. Čas, ki ga potrebujemo za inicializacijo je namreč vedno enak, saj se inicializacija izvede samo enkrat, ne glede na število korakov. Z večjim številom korakov postaja čas inicializacije v primerjavi s časom izvajanja celotnega programa čedalje manjši, pohitritev pa večja.



Slika 5.2: Graf časov izvajanja pri različnem številu korakov. Časi CPU so zaradi ogromne razlike s časi GPU pomnoženi s faktorjem 0,1.

Zanimalo nas je tudi kakšni so časi izvajanja in pohitritve pri različnih velikostih vhodnih poravnav več zaporedij. Spreminjamo lahko 2 parametra. Prvi je dolžina zaporedij aminokislinskih ostankov in je v tabelah v nadaljevanju označen z L. Ta vpliva na število možnih kombinacij mest, za katere je potrebno v vsakem koraku izračunati vrednost χ^2. Ker smo pri izvajanju meritev program vedno zagnali z dvema enako velikima poravnavama, je vseh možnih kombinacij L^2. Drugi parameter je število zaporedij v poravnavi, ki smo ga v tabelah označili z M. Ta vpliva na čas, ki ga potrebujemo za izračune ene vrednosti χ^2. Ker smo ugotovili, da čas izvajanja z večanjem števila korakov narašča dokaj enakomerno (slika 5.2), smo vse nadaljnje meritve izvajali pri 100 korakih. V nasprotnem primeru bi namreč nekatere meritve na CPU pri velikih poravnavah trajale več ur.

Izvajalne čase in pohitritve pri različnih velikostih vhodnih poravnav lahko vidimo v tabelah 5.2, 5.3, 5.4 in 5.5. Rezultati se skladajo z našimi pričakovanji, saj je paralelna implementacija ponovno precej hitrejša od sekvenčne. Na grafih na sliki 5.3 lahko vidimo, da izvajalni časi z večanjem števila aminokislinskih ostankov v zaporedju naraščajo eksponentno. To je povsem razumljivo, saj število možnih kombinacij mest narašča z L^2. Čas izvajanja seveda narašča tudi z večanjem števila zaporedij, saj se s tem poveča čas, potreben za izračun ene vrednosti χ^2. Poveča se tudi čas, potreben za mešanje enega mesta.

Na sliki 5.4 lahko vidimo, kako se spreminjajo pohitritve pri različnih velikostih vhodnih poravnav. Pri najmanjši poravnavi (M = 500, L = 250) smo na GPU dosegli 19-kratno pohitritev, pri največji poravnavi (M = 2000, L = 1500) pa kar 110-kratno pohitritev. Pri večjih poravnavah so pohitritve pričakovano večje, kot pri manjših. Večja kot je namreč poravnava, več vzporednih niti zaženemo na GPU. Zaradi tega je visoko paralelna arhitektura grafične procesne enote bolj izkoriščena. To pa seveda ne gre v nedogled, saj prej ali slej zasičimo izvajalne enote in pohitritve se lahko celo zmanjšajo. Glede na ugotovitve iz tabele 5.1, bi z večjim številom korakov pri večjih poravnavah verjetno dosegli še večje pohitritve.

L	t_{CPU} [s]	t_{GPU} [s]	S
250	31,2	1,6	19,3
500	36,0	1,6	22,2
750	65,2	2,1	30,5
1000	90,9	2,6	34,9
1250	161,4	3,2	50,1
1500	227,5	4,0	56,5

Tabela 5.2: Izvajalni časi in pohitritve pri $M = 500$ in 100 korakih.

L	t_{CPU} [s]	t_{GPU} [s]	S
250	72,2	2,2	33,3
500	112,0	2,5	45,5
750	255,3	4,4	58,0
1000	378,1	5,1	74,1
1250	628,4	7,4	84,8
1500	955,7	10,5	90,8

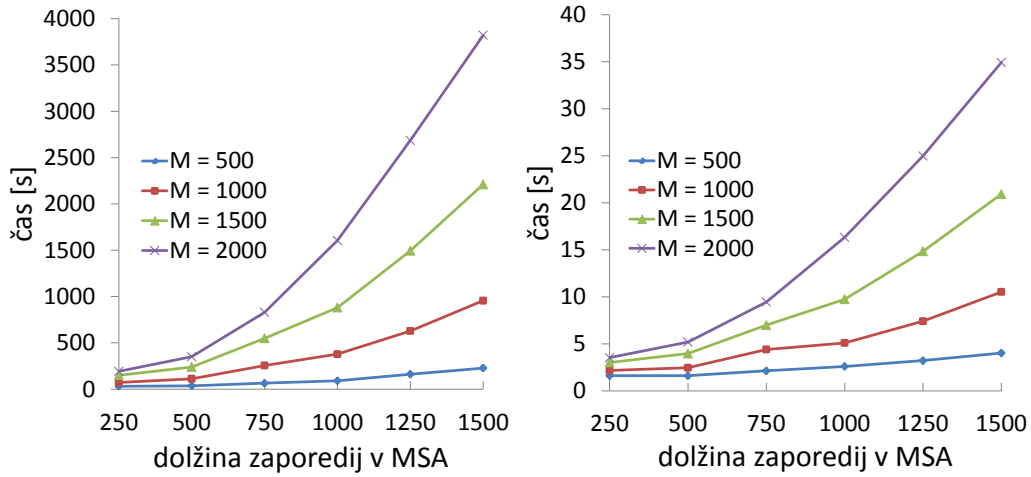
Tabela 5.3: Izvajalni časi in pohitritve pri $M = 1000$ in 100 korakih.

L	t_{CPU} [s]	t_{GPU} [s]	S
250	149,7	3,0	49,6
500	239,4	4,0	60,3
750	548,7	7,0	78,5
1000	880,2	9,7	90,4
1250	1494,0	14,8	100,7
1500	2210,6	20,9	105,7

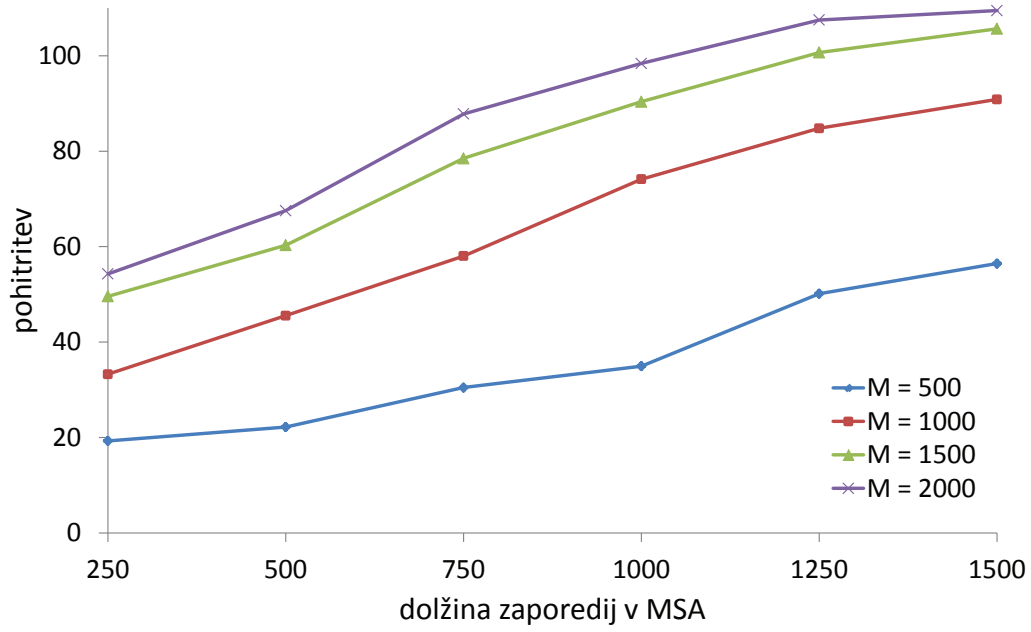
Tabela 5.4: Izvajalni časi in pohitritve pri $M = 1500$ in 100 korakih.

L	t_{CPU} [s]	t_{GPU} [s]	S
250	191,6	3,5	54,3
500	350,5	5,2	67,5
750	829,8	9,5	87,8
1000	1605,6	16,3	98,4
1250	2685,1	25,0	107,5
1500	3821,4	34,9	109,5

Tabela 5.5: Izvajalni časi in pohitritve pri in $M = 2000$ in 100 korakih.



Slika 5.3: Izvajalni časi na CPU (levo) in na GPU (desno) pri različnih velikosti MSA in 100 korakih.



Slika 5.4: Graf pohitritev pri različnih velikostih MSA.

Preverili smo še, kakšna je poraba energije med izvajanjem obeh programov. Za meritve na GPU smo uporabili orodje *NVIDIA System Management Interface*, na CPU pa *Intel Power Gadget*. Meritve smo izvajali pri 100 korakih in poravnavi več zaporedij z velikostjo $M = 2000$ in $L = 1500$. Ugotovili smo, da med izvajanjem sekvenčnega programa CPU troši moč 13 W, med izvajanjem paralelnega pa GPU troši moč 109 W. Za primerjavo, v mirovanju CPU troši moč 4,5 W, GPU pa 16 W. S hitrim izračunom ugotovimo, da je CPU za izvajanje našega programa porabil $13\,\mathrm{W} \cdot 3821\,\mathrm{s} = 13{,}8\,\mathrm{Wh}$, GPU pa $109\,\mathrm{W} \cdot 35\,\mathrm{s} = 1{,}1\,\mathrm{Wh}$ energije. Kljub temu, da grafična procesna enota pod obremenitvijo deluje s precej večjo močjo kot centralna procesna enota, je za izračun vseeno potrebovala manj energije, saj ga opravi precej hitreje. Zavedati pa se moramo da v te meritve ni vključena energija, ki jo porabijo ostale računalniške komponente.

Poglavje 6

Sklepne ugotovitve

Cilj te diplomske naloge je bila implementacija enega od algoritmov za detekcijo soodvisnih sprememb proteinov na grafični procesni enoti. V okviru dela smo najprej opisali, kaj je molekularna koevolucija in zakaj je analiza soodvisnih mutacij proteinov pomembna. Pregledali smo algoritme za detekcijo soodvisnih sprememb, in enega, algoritem OMES, podrobneje opisali. Seznanili smo se tudi z uporabo grafičnih procesnih enot za splošno-namensko računanje.

Glavni prispevek dela je zagotovo implementacija algoritma OMES na grafični procesni enoti. Sprva smo algoritem implementirali sekvenčno, na centralni procesni enoti. Nato smo ga paralelizirali in ga s pomočjo platforme CUDA implementirali še na grafični procesni enoti. Čeprav je programiranje na grafičnih procesnih enotah od izida platforme CUDA precej lažje, kot je bilo včasih, še vedno zahteva dokaj drugačen način razmišljanja kot tradicionalno sekvenčno programiranje. Kodo, ki se bo izvajala na grafični procesni enoti moramo ločiti od tiste, ki se bo izvajala na centralni procesni enoti in jo prilagoditi na izvajanje v več sočasnih nitih. Podatke moramo prenesti v pomnilnik grafične procesne enote in paziti, da do njih pravilno dostopamo. Da lahko dosežemo optimalno učinkovitost našega programa, moramo zelo podrobno poznati samo arhitekturo grafične procesne enote.

Naš program omogoča detekcijo parov mest v proteinskih in DNA/RNA

zaporedjih, na katerih lahko opazimo soodvisne spremembe. Rezultat prikaže v obliki karte proteinskih stikov. To lahko biologi in kemiki uporabijo kot vhode v mnoge druge algoritme pri raznih analizah evolucije in strukture proteinov. Naša implementacija na grafični procesni enoti je pomembna predvsem zato, ker smo v naših testih v primerjavi s sekvenčno implementacijo dosegli približno stokratne pohitritve. Na primer, čas, potreben za detekcijo soodvisnih sprememb med dvema poravnavama z 2000 zaporedji dolžine 1500 aminokislinskih ostankov, smo z več ur skrajšali na dobro minuto. Zaradi krajšega časa izvajanja je precej manjša tudi poraba energije.

Seveda pa je v našem programu še prostor za izboljšave. Ena od možnih izboljšav bi bila uporaba deljenega pomnilnika, ki ima precej nižje zakasnitve dostopa kot globalni pomnilnik. Najbolj očitno mesto, kjer bi ga lahko uporabili je mešanje poravnav več zaporedij. Tam namreč pri vsakem mešanju vsaka nit večkrat piše v globalni pomnilnik. Naslednja možna izboljšava bi bila prilagoditev na izvajanje na več grafičnih procesnih enotah hkrati. Naša implementacija trenutno podpira zgolj izvajanje na eni grafični procesni enoti.

Literatura

[1] S. C. Lovell in D. L. Robertson, "An integrated view of molecular coevolution in protein–protein interactions," *Molecular biology and evolution*, št. 27, zv. 11, str. 2567–2575, 2010.

[2] A. A. Abu-Doleh, O. M. Al-Jarrah, in A. Alkhateeb, "Protein contact map prediction using multi-stage hybrid intelligence inference systems," *Journal of biomedical informatics*, št. 45, zv. 1, str. 173–183, 2012.

[3] J. N. Thompson, *The coevolutionary process*. University of Chicago Press, 1994.

[4] M. Tokeshi, *Species coexistence: ecological and evolutionary perspectives*. John Wiley & Sons, 2009.

[5] D. H. Janzen, "When is it coevolution," *Evolution*, št. 34, zv. 3, str. 611–612, 1980.

[6] Wikipedia, "Phylogenetics," 2014. Dostopno na: http://en.wikipedia.org/w/index.php?title=Phylogenetics&oldid=617843965 (23. 7. 2014).

[7] Wikipedia, "Locus (genetics)," 2014. Dostopno na: http://en.wikipedia.org/w/index.php?title=Locus_(genetics)&oldid=617901828 (30. 7. 2014).

[8] H. Ashkenazy, R. Unger, in Y. Kliger, "Optimal data collection for correlated mutation analysis," *Proteins: Structure, Function, and Bioinformatics*, št. 74, zv. 3, str. 545–555, 2009.

[9] I. Kass in A. Horovitz, "Mapping pathways of allosteric communication in groel by analysis of correlated mutations," *Proteins: Structure, Function, and Bioinformatics*, št. 48, zv. 4, str. 611–617, 2002.

[10] H. Lodish, *Molecular cell biology*. Macmillan, 2008.

[11] Wikipedia, "Protein folding," 2015. Dostopno na: http://en.wikipedia.org/w/index.php?title=Protein_folding&oldid=642807810 (26. 1. 2015).

[12] L. S. Swapna, N. Srinivasan, D. L. Robertson, in S. C. Lovell, "The origins of the evolutionary signal used to predict protein-protein interactions," *BMC evolutionary biology*, št. 12, zv. 1, str. 238, 2012.

[13] R. C. Edgar in S. Batzoglou, "Multiple sequence alignment," *Current opinion in structural biology*, št. 16, zv. 3, str. 368–373, 2006.

[14] C. B. Do in K. Katoh, "Protein multiple sequence alignment," v *Functional Proteomics*. Springer, 2008, str. 379–413.

[15] A. E. Marquez-Chamorro, G. Asencio-Cortes, F. Divina, in J. S. Aguilar-Ruiz, "Evolutionary decision rules for predicting protein contact maps," *Pattern Analysis and Applications*, str. 1–13, 2012.

[16] D. R. Livesay, K. E. Kreth, in A. A. Fodor, "A critical evaluation of correlated mutation algorithms and coevolution within allosteric mechanisms," v *Allostery*. Springer, 2012, str. 385–398.

[17] D. de Juan, F. Pazos, in A. Valencia, "Emerging methods in protein coevolution," *Nature Reviews Genetics*, št. 14, zv. 4, str. 249–261, 2013.

[18] K. Y. Yip, P. Patel, P. M. Kim, D. M. Engelman, D. McDermott, in M. Gerstein, "An integrated system for studying residue coevolution in proteins," *Bioinformatics*, št. 24, zv. 2, str. 290–292, 2008.

[19] I. Halperin, H. Wolfson, in R. Nussinov, "Correlated mutations: Advances and limitations. a study on fusion proteins and on the cohesin-

dockerin families," *Proteins: Structure, Function, and Bioinformatics*, št. 63, zv. 4, str. 832–845, 2006.

[20] U. Göbel, C. Sander, R. Schneider, in A. Valencia, "Correlated mutations and residue contacts in proteins," *Proteins: Structure, Function, and Bioinformatics*, št. 18, zv. 4, str. 309–317, 1994.

[21] A. A. Fodor in R. W. Aldrich, "Influence of conservation on calculations of amino acid covariance in multiple sequence alignments," *Proteins: Structure, Function, and Bioinformatics*, št. 56, zv. 2, str. 211–221, 2004.

[22] V. Baths in U. Roy, "Identification of distant co-evolving residues in antigen 85c from *Mycobacterium tuberculosis* using statistical coupling analysis of the esterase family proteins," *Journal of biomedical research*, št. 25, zv. 3, str. 165–169, 2011.

[23] O. Noivirt, M. Eisenstein, in A. Horovitz, "Detection and reduction of evolutionary noise in correlated mutation analysis," *Protein Engineering Design and Selection*, št. 18, zv. 5, str. 247–253, 2005.

[24] Wikipedia, "P-value," 2014. Dostopno na: http://en.wikipedia.org/w/index.php?title=P-value&oldid=618791870 (15. 8. 2014).

[25] nVidia, "Cuda c programming guide," 2014. Dostopno na: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (15. 1. 2015).

[26] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, in J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, št. 96, zv. 5, str. 879–899, 2008.

[27] T. D. Han in T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *Parallel and Distributed Systems, IEEE Transactions on*, št. 22, zv. 1, str. 78–90, 2011.

[28] J. Sanders in E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[29] J. Nickolls in W. J. Dally, "The gpu computing era," *IEEE micro*, št. 30, zv. 2, str. 56–69, 2010.

[30] D. B. Kirk in W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[31] Wikipedia, "Cuda," 2014. Dostopno na: http://en.wikipedia.org/w/index.php?title=CUDA&oldid=641728337 (17. 9. 2014).

[32] D. Kodek, *Arhitektura in organizacija računalniških sistemov*. Bi-tim, 2008.

[33] "1000 genomes project," 2014. Dostopno na: http://www.1000genomes.org/ (21. 1. 2015).